

Introduction to Programming Concepts and Tools

Lecture 8:

Dynamic Data Structures

Pohl & McDowell: Java by Dissection.

Chp. 12.1-12.5, 12.7, 12.10, 12.11

Today's Lecture:

- Dynamic data structures
 - Linked List
 - Stack
 - Queue
- Generic data structures
 - generic stack
- Iterators
- Packages
- Binary Tree

Dynamic data structures
Recursive data structures
Self referential data structures

```
class IntListElement{  
    int data;  
    IntListElement next;  
}
```

Dynamic Data Structures

- dynamic: the size can be changed as needed

an array has a fixed size,

elements can be accessed directly

a linked list has dynamic size,

elements are accessed via pointers,

elements are accessed via pointers to (objects with) pointers

elements are accessed via pointers to (objects with) pointers to pointers

dynamic data structures

When you program with dynamic data structures use drawings as a help in developing methods

Be sure to keep track of all pointers

An element that you cannot reach via pointers
is not accessible for you any more

a singly linked list

```
class IntListElement{  
    int data;  
    IntListElement next;}
```

- rather than building a list only using this class, we will define one more `class IntList{..}`
 - we can guarantee certain properties of **IntLists**
 - we can have data fields that facilitates the use we expect
 - we can define methods once for list operations, and ensure that they preserve the list properties we want

singly linked list

```
class IntListElement{
    int data;
    IntListElement next;
    IntListElement(int value){
        data = value;
    }
}
```

a singly linked list

```
class IntList {  
    private IntListElement head, current, previous;  
  
    void insert(int value) {...}  
        //inserts a new element after current, updates current  
  
    int current() {...}  
        // returns the data from current element  
  
    void remove() {...}  
        //removes current element, keeps list connected  
  
    void moveToHead() {...}  
        //places current before head (current=null)  
  
    boolean hasNext() {...}  
        //checks if current has next element  
  
    int next() {... }  
        //moves current to next, exception if list is empty  
}
```

a singly linked list

Seen from outside the class

when we have a variable of type IntList:

- our view is always positioned at some element (current)
- we can read the data field of the current element
- we can change the element we view
 - move current to next, -to head
- we can insert elements specifying their **data** only
- we can remove elements

- All internal pointer manipulations when list is modified are hidden for our view

a singly linked list

An IntList can only be modified via the methods

```
void insert(int) and void remove()
```

```
// but a variable can be modified to point to something else
```

- All internal pointer manipulations when list is modified are hidden for our view
- If `insert(int)` and `remove()` prevents that there comes a loop in the list, then this is ensured (and can be assumed by the other methods)

a singly linked list

```
class IntList {  
    private IntListElement head, current, previous;  
  
    void insert(int value) {...}  
        //inserts a new element after current, updates current  
  
    int current() {...}  
        // returns the data from current element  
  
    void remove() {...}  
        //removes current element, keeps list connected  
  
    void moveToHead() {...}  
        //places current before head (current=null)  
  
    boolean hasNext() {...}  
        //checks if current has next element  
  
    int next() {... }  
        //moves current to next, exception if list is empty  
}
```

```
void moveToHead() {  
    previous = current = null;  
}
```

```
boolean hasNext() {  
    if (current != null)  
        return current.next != null;  
    else  
        return head != null;  
}
```

```
int next() {  
    previous = current;  
    if (current == null)  
        current = head;  
    else  
        current = current.next;  
    return current.data;  
}
```

what should you check before you use this method?

```

void insert(int value) {
    previous = current;
    if (current != null) {
        IntListElement next = current.next;        //step 1
        current.next = new IntListElement(value); //step 2
        current = current.next;                    //step 3
        current.next = next;                       //step 4
    }
    else if (head != null) {
        current = new IntListElement(value);
        current.next = head;
        head = current;
    }
    else /* list is empty */
        head = current = new IntListElement(value);
}

```

```
void remove() {  
    if (head == null) return; //list is empty  
    if (current == null) return;  
    if (current == head)  
        head = current = current.next;  
    else  
        current = previous.next = current.next;  
}
```

when an element has been removed, we have no pointers to it any more

doubly linked list

```
class IntDListElement{
    int data;
    IntDListElement next;
    IntDListElement previous;
    IntDListElement(int value){
        data = value;
    }
}
```

Stacks and Queues

- dynamic sets in which the element removed from the set by a delete operation is pre-specified
 - Stack: LIFO LAST IN FIRST OUT
the element removed is the one most recently inserted
 - Queue: FIFO FIRST IN FIRST OUT
the element removed is the one that has been in the queue for the longest time

Stack of integers

what can we do with a stack ?

- check if the stack is empty
- view the top
- pop off the top
- push an element unto the top

Stack of integers

we can use `IntListElement` as our element-type

```
-----  
class IntStack {  
    private IntListElement top = null;  
  
    boolean empty() {...}  
  
    int top() {...}  
        //return top.data, exception if stack is empty  
  
    void push(int value) {...}  
        //push element with data = value on top  
  
    int pop() {...}  
        //calls top() method and resets top pointer
```

Stack of integers

If we try to access a field or method for a pointer that is null, for instance if `int top() {return top.data;}` is called for an empty stack, then we will get a **NullPointerException**.

At the lecture we discussed that if we know that we only want to keep certain values in our stack, then we can return an illegal value if `int top()` is called with an empty stack.

```
int top() {..if not empty return data, else return -1..}
```

(Can we ensure that we always insert legal values?)

Queue of integers

what can we do with a queue ?

- check if the queue is empty
- view the head of the queue
- remove the head of the queue
- insert an element (after the last element)

Queue of integers

we can use `IntListElement` as our element-type

```
class IntQueue {  
    private IntListElement head = null;  
    private IntListElement last = null;  
  
    boolean empty() {...}  
  
    Int front() {...}  
        // return data, exception if empty  
  
    void add(int value) {...}  
        // add an element after the last  
  
    Int pop() {...}  
        // calls front() and removes head  
}
```

The pointer manipulations in the implementations of `IntStack` and `IntQueue` are less involved than for `IntList` because all modifications are in the front or in the end

```
class Queue {
    private IntListElement head = null;
    private IntListElement last = null;
    boolean empty() { return head == null; }

    void add(Int value) {
        if (head == null) {
            head = last = new IntListElement(value);
        }
        else {
            last.next = new IntListElement(value);
            last = last.next;
        }
    }
    .....
}
```

.....

```
Int front() { // exception if called with an empty queue
    return head.data;
}
```

```
Int pop() { // if front ( ) can give an exception then this can
            // we could define front ( ) differently
    Int result = front();
    if (head != null)
        head = head.next;
    return result;
}
}
```

The ADT Stack in Java

The following is an implementation of a generic stack in Java.

In this stack implementation we also keep track of how many elements the stack has

```

public class Stack {
    private StackElement top;
    private int count=0;
    public void push(Object obj){
        StackElement element = new StackElement(obj);
        element.next = top;
        top = element;
        count++;
    }
    public Object pop() {
        if(top == null) return null;
        Object obj = top.value;
        top = top.next;
        count--;
        return obj;
    }
}

```

the method top() is omitted. We could easily give the size of the stack

```
class StackElement {
    Object value;
    StackElement next;
    StackElement(Object obj)
    {
        value = obj;
        next = null;
    }
}
```

When a program that use a generic stack want to retrieve the data from the elements it must know the actual type of the value (to do something with it).

Then it can use a cast from Object to the relevant type

```
Integer x = (Integer)myStack.pop( )
```

```

public class StackTest {
    public static void main(String[] args)
    {
        Stack stack = new Stack();
        stack.push("Testing");
        stack.push("One");
        stack.push(new Integer(99));
        System.out.println(stack.pop());
        System.out.println(stack.pop());
        System.out.println(stack.pop());
        System.out.println(stack.pop());
    }
}

```

-----output:

99

One

Testing

null

new Integer(99)

wrapper class

wraps a primitive **int**

inside an object

Iterators

- allows you to iterate over a collection of objects (array, list, tree)
- it must be possible to advance an iterator to the next element in the collection (in a specified way)
- it must be possible to determine if you have iterated over all elements
(or to move to the beginning and to recognize if the end is reached)

Iterators

- an iterator for an array can be a variable `int i` used to index into the array
- an iterator for a linked list ?
 - the methods `hasNext()` and `next()` gives you one implicit iterator, but it is not possible to use these methods to get two different positioned iterators

Iterator for a linked list

- An iterator will need to know the type of the elements and be able to see them
- In general an iterator will need to know some internal details of implementation
- Solution. We let the class implementing the linked list handle the creation of the iterator

Iterator for a linked list

// we add a method for creating an iterator

// to the class IntList

```
class IntList {
```

```
    ...
```

```
    IntListIterator iterator(){
```

```
        return new IntListIterator(this);
```

```
    ...
```

```
}
```

// the iterator should be able to traverse the list positioned

// differently than the current-pointer that belong to the list

Iterator for a linked list

// defined in the same package as the class IntList

```
class IntListIterator {
```

```
    IntListElement curr;
```

```
    IntList list;
```

// the constructors initializes curr and list

```
    IntListIterator(IntList list){
```

```
        curr = null;
```

```
        this.list = list    }
```

```
    IntListIterator(IntList list, IntListElement pos){
```

```
        curr = pos;
```

```
        this.list = list    }
```

// methods...next slide

```
}
```

Iterator for a linked list

```
class IntListIterator {  
    IntListElement curr;  
    IntList list; .....  
  
    int next() { moves curr to the next element,  
                returns the data. Exception if null }  
  
    boolean hasNext() {checks if there is an element after curr }  
  
    int curr () {return curr.data;}  
}
```

these methods are implemented similarly to the methods in IntList
but to refer to the head we must have `list.head`

Deleting Objects

Garbage Collection

- we create objects with the keyword **new**
Each object occupies some space in memory
- java has **garbage collection**, it is automatically figured out if an object is no longer needed, then the memory-space can be reused.
This happens when there is no way an object can be referenced.

Packages

- Some details of implementation can be accessible for other classes in the same package, but *not* for classes outside the package
- Packages provides separate name spaces
- Package access is default when no access-modifier is specified. If no package is specified then classes in the same directory is considered as constituting a package

Packages

- Reuse of code. A program can use classes that were written (and thoroughly tested) by others.
- Possible to specify package access to whole classes as well as to individual fields and methods
- Classes, methods and fields that should be visible outside the package must be specified as public

public/private/default

- **private** methods/fields cannot be accessed from outside of the class
- **public** methods/fields can be accessed from anywhere
- default (no modifier) methods/fields have package access. They can be accessed from other classes in the same package.
 - If you don't specify a package, all classes in the same directory are part of the same, default - unnamed package.

Creating a package

- add a package statement to the top of each class in the package

```
//intListElement.java  
package intContainers;  
  
.....
```

- place the classes from the package `intContainers` in a subdirectory of the same name: `intContainers`

compiling

you can compile all .java files in a package at once.

Execute the following command

from the directory

where the package-name-directory is a subdirectory

either `javac intContainers*.java`

or `javac intContainers/*.java`

Rooted Tree

a rooted tree is a connected, acyclic, undirected graph with a distinguished node

root

ancestors of a node x

descendants of a node x

parent of a node x

children of a node x

internal node

leaf

subtree

Binary Tree

In a binary tree each node has at most two children

Recursive description: A binary tree is a structure defined on a finite set of nodes that either

- contains no nodes (is empty)

OR

- consist of :

 - a root node

 - the left subtree, a binary tree,

 - the right subtree, a binary tree

Binary Tree

- complete tree
- height of a tree

- tree walk
 - inorder: a node is taken between its two children
 - preorder: a node is taken before its children
 - postorder. a node is taken after its children

Binary Tree

Can be implemented as a linked data structure
each node is an object holding:

key,	the data,
left	reference
right	reference
(parent	reference)

heap

- leftist binary tree satisfying the heap condition
 - heap condition: the value at each node is \leq the value at each of its children
- operations: delete minimum,-- insert,--
- understand the idea (draw)
 - understand how references are used

Binary Search Tree

Can be implemented as a linked data structure

each node is an object holding:

key,	the data, there must be an ordering on the dataset
left	pointer
right	pointer
parent	pointer

binary search tree property:

For any node x in the tree it holds that

if y is a node in the left subtree then $y.key \leq x.key$

if z is a node in the right subtree then $z.key \geq x.key$

Binary Search Tree

some operations on binary search trees:

search(key)

maximum

minimum

print in sorted order – *which tree walk method should we use?*

insert

delete

(balancing)

treeSearchRec(x, k)	<i>recursive definition of search method</i>
if x = null or x.key = k	<i>pseudo code</i>
return x	<i>x is a reference to a tree/node</i>
if k < x.key	<i>k is a value</i>
return treeSearchRec(x.left,k)	
else	
return treeSearchRec(x.right,k)	

treeSearchIt(x, k)	<i>iterative definition of search method</i>
while x != null and x.key != k	
if k < x.key	
x = x.left	
else	
x = x.right	
return x	

structure preserving operations

- A dynamic data structure supports certain operations. Operations on dynamic data structures should preserve the properties of the structure such that the operations repeatedly can be performed correctly.
 - list operations should preserve the linear structure of the list
 - heap operations should preserve the heap property as well as the leftist tree structure.

java object oriented programming is well suited to ensure that structure is preserved by operations⁵⁰