# Introduction to ds

## Data Structures for Games

Michael Baczynski
polygonal.de

# Overview

## Part 1 – Preface

- About *ds*, design goals, principles and features

## Part 2 – The Data Structures

- More detailed description of the included data structures

## Part 3 – The Collection Interface

- The interface implemented by all data structures

# Preface

# What is ds?

A haXe library providing basic data structures

Created for game programmers, not computer scientists

Simple – does not compete with C++ STL or Java collections, yet covers most of the programmer's daily needs

A learning project

Project hosting on Google Code

↳ http://code.google.com/p/polygonal

Documentation

↳ http://www.polygonal.de/doc/ds

Questions, comments, feature requests …

↳ https://groups.google.com/group/polygonal-ds

# Why ds?

Free and open source (non-restrictive BSD license)

Saves you hours of coding – game development is hard enough!

Well supported & maintained

Optimized from the ground up for AVM2

Pre-compiled SWC libraries for ActionScript 3.0 available

- ↳ http://code.google.com/p/polygonal/wiki/UsingActionScript3

# What is haXe?

HaXe is high-level language developed by Nicolas Canasse

Syntax similar to ActionScript and Java

Cross-platform – Flash, JavaScript, PHP, C++, Neko, C#, Java

Tons of features – iterators, typedefs, generics, macros …

Homepage

- ↳ http://haxe.org/

More

- ↳ http://ncannasse.fr/file/FGS2010_haxe4GD.pdf
- ↳ http://ui.massive.com.au/talks/

# Why haXe?

## Supports type parameters – no more dynamic containers

```
class Container<T> {
    var data:Array<T>;
}
var container = new Container<String>();
```

## Supports iterators – less-boilerplate code

```
for (element in myContainer) { … }
```

## Type inference – don't repeat yourself

```
var i = 3; //typed as integer
i = "3";   //compile error: String should be Int
```

## Performance – clever compiler optimizations

- Better byte code, function inlining, constant expression optimization …

## … perfect match for writing data structures!

# History

| | |
|---|---|
| 2006 | Wrote some basic data structures in ActionScript 2.0 |
| 2007 | Switched to ActionScript 3.0 |
| 2008 | Released "AS3 Data Structures for Game Developers" (*as3ds*) |
| 2009 | Switched to the haXe language |
| 2010 | Released *ds*, an improved version of *as3ds* |
| | ↳ http://lab.polygonal.de/?p=961 |

# Design Goals

## Reasonable small API

- Short learning curve
- Keep number of interfaces small
  - One “container” type (Collection<T>)
  - One iterator type

## Performance oriented

- Efficient data structures lead to efficient programs
- Fun to push boundaries

## Improve development cycle

- Human-readable error messages
- Assertions

# What Are Data Structures?

A way of storing and organizing data in a computer

A data structure includes …

1) A set of operations
2) A storage organization of the data
3) Algorithms that manipulate the data through 1)

Examples

- Primitives, e.g. the built-in integer data type
- Arrays – a sequence of data items of the same type
- Objects – a bunch of objects of various types

# Abstract Data Type – ADT

## An ADT specifies a data type & a defined set of operations

- No implementation details are given → the “logical” level
- Requires a „concrete” data structure → the implementation level

## There are many ways to implement ADTs

- Only allowed difference is performance characteristic
  - How does the run time change as the number of items increases?

## ADTs in *ds*

- Stack<T>, Queue<T>, Deque<T>, Map<K,T>, Set<T>

## Example

- Stacks can be implemented by using arrays or linked lists
- The behavior of a stack is an ADT
- Both implementations are different data structures

# Abstract Data Type – ADT (cont.)

## Objective

- Reduce complexity between algorithms & data structures
- Hide implementation details – principle of encapsulation
- Provide a higher-level abstraction of the problem

## Benefits

- Easier to understand
- Easier to organize large programs
- More convenient to change
- Less bugs!

# Features (version 1.35)

2D-, 3D-array

Singly-, Doubly-Linked Lists

Stack, Queue, Deque

Set, Map

Multiway Tree, Binary Tree, Binary Search Tree (BST)

Heap, Priority Queue

Graph

Bit vector

# Features (cont.)

All structures are of varying length (dynamic)

Arrayed & linked implementations

Iterative & recursive traversal algorithms

Debug build with additional assertions & check routines

Code performance

Object pooling helpers

Memory manager for fast virtual memory ("alchemy")

# Dynamic Data Structures

All structures in *ds* are dynamic

- A static structure has a fixed size whereas a dynamic structure automatically grows & shrinks with demand

Flash does not release memory of shrunken arrays

- Setting the length property of an array to zero has no effect
- To release memory, it's required to create a smaller array and copy the data over
- Arrayed structures in *ds* do this automatically for the user by calling Collection.pack() or Collection.clear(true)

Some collections can be made non-resizable to prevent frequent & expensive resizing if the target size is known in advance

# Arrayed v Linked

*ds* includes arrayed and linked versions of many data structures

## Arrayed – pros and cons

- Random access in constant time
- Compact, but small arrays waste memory since allocation is done in chunks
- Modifying array elements is expensive → movement of data
- Poor Flash performance

## Linked – pros and cons

- Random access in linear time
- Fast insertion & deletion by adjusting pointers
- Implicit resizing performed by insertion/removal algorithms
- Adds storage overhead per element
- Requires bookkeeping of pointers that hold the structure together
- Excellent Flash performance

# Iterative v Recursive

Some methods in *ds* can be invoked in a recursive or iterative manner

## Iterative – pros and cons

- Fast for small algorithms → allows function inlining
- Implementation is usually more complex
- Requires a helper structure (e.g. a stack or a queue)

## Recursive – pros and cons

- Implicit use of the call stack → easier to implement, fewer lines of code
- Generally slower due to overhead of maintaining call stack and function calls
- Big data sets can trigger a stack overflow due to deep recursion

# Iterative v Recursive Example

## Example – printing all elements of a linked list

### Iterative version

```
var node = head;
while (node != null) {
    trace(node);
    node = node.next;
}
```

### Recursive version – roughly 3x slower in Flash

```
function print(node) {
    if (node == null) return;
    trace(node);
    print(node.next);
}
print(head);
```

# Debug v Release Build

In *ds*, debug-builds behave differently than release-builds

Debug build

- Validates user input (e.g. index out of range)
- Provide meaningful error messages
- Catch errors early!

Release build

- Includes only the bare minimum parts for best performance
- Silently fails if something goes wrong!
- Even allows illegal operations that renders the structure useless!

Always use the debug version during development

- Using haXe, compile with -debug directive
- Using ActionScript, compile against ds_debug.swc

# Debug v Release Example 1

## Example – popping data of an empty array silently fails in Flash

### Using a flash array

```
var stack = new Array<Int>();
stack.push(0);
stack.pop();
stack.pop(); //stack underflow
```

### Using an ArrayedStack object in debug mode

```
var stack = new de.polygonal.ds.ArrayedStack<Int>();
stack.push(0);
stack.pop();
stack.pop(); //throws: Assertation 'stack is empty' failed
```

# Debug v Release Example 2

The “denseness” of a dense array is only checked in debug mode – boundary checking every access is expensive!

Example – adding elements to a dense array

## Release

```
var da = new de.polygonal.ds.DA<Int>();
da.set(1, 100); //array is no longer dense!
```

## Debug

```
var da = new de.polygonal.ds.DA<Int>();
da.set(1, 100); //throws 'the index 1 is out of range 0' failed
```

# Debug v Release Example 3

Some operations render a structure useless when used in certain conditions

Example – adding an element to a fixed-size, full queue

### Prerequisite

```
var isResizable = false;
var maxSize = 16;
var que = new de.polygonal.ds.ArrayedQueue<Int>(maxSize, isResizable);
for (i in 0...maxSize) {
    que.enqueue(i); //fill the queue
}
```

### Release

```
que.enqueue(100); //silently overwrites an existing item!
```

### Debug

```
que.enqueue(100); //throws: Assertion 'queue is full' failed
```

# Performance Guidelines

## Favor code efficiency over utilization efficiency

- It's far more efficient to find a dedicated, specialized method instead of re-using and recombining existing methods

## Favor interfaces over functions literals

- Much faster for strictly typed runtimes (Flash, C++, Java, C#)
- Typed function calls are almost 10x faster in AVM2

## Use non-allocating implementations

- Prevent frequent allocation of short-lived objects that need to be GCed
- Node based structures offer built-in object pooling

## Prefer composition over inheritance

- Avoid slow casts where possible

# Performance – Comparing Elements

## Example – comparing elements using an interface (faster)

### Prerequisite

```
class Foo implements de.polygonal.ds.Comparable<Foo> {
    public var val:Int;
    public function new() {}
    public function compare(other:Foo):Int { return val - other.val; }
}
```

### Usage

```
myFoo.compare(otherFoo);
```

## Example – comparing elements using a function literal (slower)

```
var compare = function(a:Foo, b:Foo) { return a.val - b.val; }
compare(myFoo, otherFoo);
```

## User choice!

# Performance – Reusing Objects

Pass objects to methods for storing their output to prevent object allocation inside methods

Example – extracting a row from a 2-dimensional array

```
var matrix = new de.polygonal.ds.Array2<Int>(10, 10);
var output = new Array<Int>(); //stores the result

matrix.getRow(0, output); //output argument stores row at y=0
matrix.getRow(1, output); //reuse output to store another row
…
```

# Object Pooling

Manages a set of pre-initialized objects ready to use

Avoids objects being allocated & destroyed repeatedly

Significant performance boost when …

- Class instantiation is costly
- Class instantiation is frequent
- Instantiated objects have a short life span

Performance-memory trade-off

# Object Pooling Implementation

## ObjectPool<T>

- A fixed-sized, arrayed object pool implemented as a “free list” data structure
- Objects are accessed by integer keys
- Requires to keep track of the key, not the object itself
- Object can be initialized on-the-fly (lazy allocation) or in advance

## DyamicObjectPool<T>

- A dynamic, arrayed object pool implemented as a stack
- Pool is initially empty and grows automatically
- If size exceeds a predefined limit a non-pooled object is created on-the-fly
  - Slower, but application continues to work as expected

# Object Pooling Example

## Example – using an ObjectPool

```
import de.polygonal.ds.pooling.ObjectPool;

var capacity = 1000;
var pool = new ObjectPool<Foo>(capacity);

var objects = new Array<Int>();
for (i in 0...10) {
    var key = pool.next(); //get next free object key from the pool
    objects.push(key);     //keep track of those keys for later use
}

for (key in objects) {
    var foo:Foo = pool.get(key); //key -> object
    foo.doSomething();
    pool.put(key); //return object to the pool
}
```

# Alchemy Memory
# †2011*

*Flash Player 11.2 will not support the experimental Alchemy prototype

Adobe Make Some Alchemy !
http://ncannasse.fr/blog/adobe_make_some_alchemy

# Fast Alchemy Memory

Alchemy toolchain transforms C/C++ into ActionScript bytecode

ByteArray objects are too slow for the C memory model so Adobe added special opcodes for fast memory access

haXe exposes those opcodes through a simple memory API (flash.memory.*)

## Example

```
import flash.utils.ByteArray;
var bytes = new ByteArray(4096); //create 4 KiB of memory
flash.Memory.select(bytes);      //make bytes accessible through memory api
flash.Memory.getI32(i);          //read 32-bit integer from byte address i
flash.Memory.setI32(i, x);       //write 32-bit integer x to address i
```

## More

- ↳ http://ncannasse.fr/blog/virtual_memory_api
- ↳ http://labs.adobe.com/wiki/index.php/Alchemy:FAQ

# Fast Alchemy Memory (cont.)

## Idea

- Create super fast arrays for number crunching with a simple API

## Naïve solution

- Use multiple ByteArray objects – each one representing an array object
- Call flash.Memory.select() before accessing it

## Problem

- Calls to flash.Memory.select() are too expensive

## Solution

- Split a single ByteArray object into smaller pieces → chunks of memory
- The ByteArray is managed by a dynamic memory allocator
  - de.polygonal.ds.MemoryManager

# MemoryManager

## Allocating memory

`MemoryManager.malloc(accessor:MemoryAccess, numBytes:Int):Void`

- Finds a block of unused memory of sufficient size (using “first fit” allocation)
- A chunk of memory is represented by a MemorySegment object
- Configures accessor parameter to point to the segment's address space

## Deallocating memory

`MemoryManager.dealloc(accessor:MemoryAccess):Void`

- Returns used bytes to the memory pool for later use by the program
- By default, memory isn't automatically reclaimed
  - User has to call MemoryAccess.free() in order to prevent a memory leak
  - If MemoryManager.AUTO_RECLAIM_MEMORY is true,
    memory is automatically reclaimed when an object
    extending MemoryAccess is GCed (using weak reference hack)

# MemoryManager (cont.)

## Classes using virtual memory (de.polygonal.ds.mem.*)

| | |
|---|---|
| BitMemory | Array storing bits (“bit vector”) |
| ByteMemory | Array storing bytes (fast ByteArray replacement) |
| ShortMemory | Array storing signed 16-bit integers |
| IntMemory | Array storing signed 32-bit integers |
| FloatMemory | Array storing 32-bit floating point numbers |
| DoubleMemory | Array storing 64-bit floating point numbers |

## Cross-platform compatibility

- Supported in Flash and C++ target
- Alchemy opcodes are only used when compiled with -D alchemy
- If omitted, flash.Vector is used as a fallback

## More

↳ http://lab.polygonal.de/?p=1230

# MemoryManager Example

## Example – basic usage

```
import de.polygonal.ds.mem.IntMemory;

var memory = new IntMemory(100); //allocates space for 100 integers
memory.set(4, 10);               //store value 10 at integer index 4
var x = memory.get(4);           //return value at index 4
memory.free();                   //deallocate once no longer needed
```

## Example – fast iteration

```
var memory = new IntMemory(100);
var offset = memory.offset; //byte offset of this memory segment
for (i in 0...100) {
    //integer index = byte index * 4
    var x = flash.Memory.getI32(offset + i << 2);
}
```

# The Data Structures

# Multi-Dimensional Arrays

Includes a two- and three-dimensional array

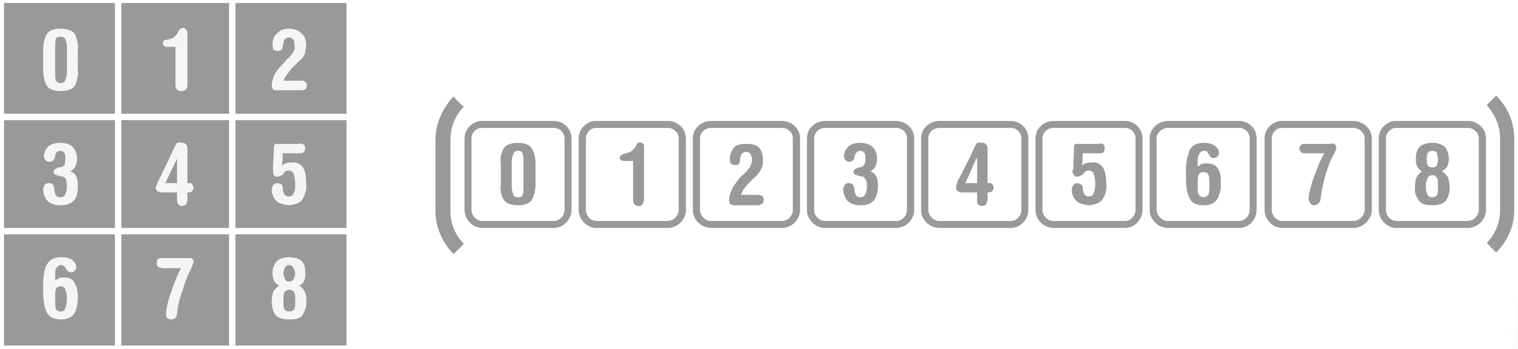
Elements are stored in a rectangular sequential array

- Rows are laid out sequentially in memory
- Row-major order – kind of C/C++ creates by default
- 2D array index: (y * width) + x
- 3D array index: (z * width * height) + (y * width) + x

Fast – only one array access [ ] operation in any dimension

Dense – efficient memory usage

- Array locations for a 3x3 matrix, stored sequentially:



| | | |
|---|---|---|
| 0 | 1 | 2 |
| 3 | 4 | 5 |
| 6 | 7 | 8 |

( 0 1 2 3 4 5 6 7 8 )

# Linked Lists

## Several objects (“nodes”) linked together

- A node stores a value (“cargo”) and a reference to the next (& previous) node
- Nodes can be rearranged and added/removed efficiently
- In *ds*, nodes are managed by a list class

## Features

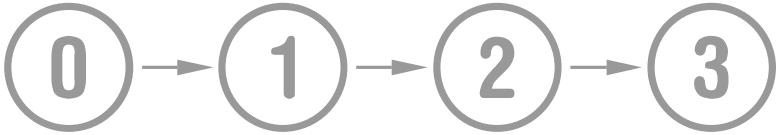
- Supports mergesort & insertionsort – latter is very fast for nearly sorted lists
- Supports circular lists
- Built-in node pooling to avoid node allocation (optional)

## More (based on *as3ds*)

↳ http://lab.polygonal.de/?p=206

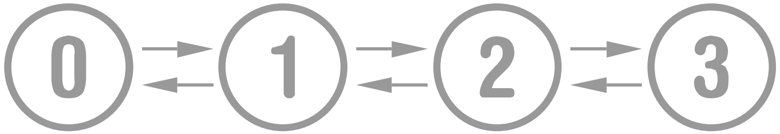
# Singly v Doubly Linked Lists

## Singly linked list (de.polygonal.ds.SLL<T>)



- Can't traverse list backwards
- Can't delete item only given a reference to that node → removal takes linear time
- Overhead: 4 extra bytes per node in Flash (reference to next node)

## Doubly linked list (de.polygonal.ds.DLL<T>)



- Can be traversed either forward or backward
- Removal of elements in constant time
- Overhead: 8 extra bytes per node in Flash (reference to next & previous node)

# Circular Linked Lists

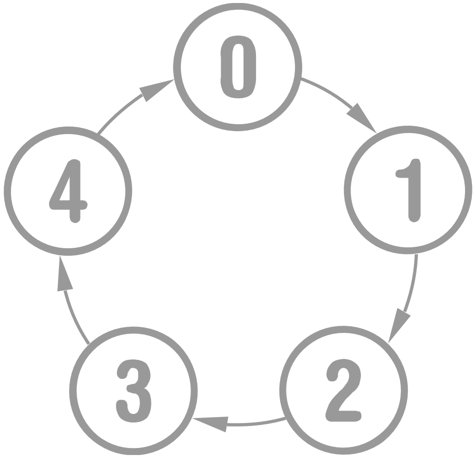
A linked list is linearly-linked (“open”) by default

A linked list can be transformed into a circular-linked list with myList.close()

When closed, null is no longer used to terminate the list –
instead the tail points to the head (and v.v. for doubly-linked lists)

Iterating over a circular linked list can result in an infinite loop:

```
var node = myList.head;
while (node != null) {
    if (node == myList.tail) { //check end condition!
        break;
    }
    node = node.next;
}
```



# Linked List Example

## Example – fast self-removal of list elements by cross-referencing

## Prerequisite

```
class Foo {
    public var node:de.polygonal.ds.DLLNode<Foo>;
    public function new() {}
    public function remove():Void {
        node.unlink();
        node = null;
    }
}
```

## Usage

```
var list = new de.polygonal.ds.DLL<Foo>();
var foo = new Foo();
foo.node = list.append(foo);
…
foo.remove(); //remove foo from list
```

# Destroying a Linked List

**It's sufficient to drop the head of the list because the garbage collector finds and reclaims all remaining nodes ...**

```
head = null;
```

**... but nullifying all references improves garbage collection**

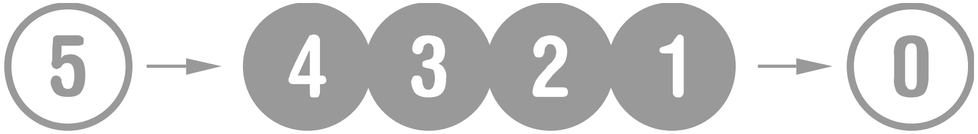
```
var node = head;
while (node != null) {
    var hook = next;  //don't fall of the list
    node.next = null; //nullify pointer
    node = hook;
}
```

**Applied by *ds* to all node-based collections when calling Collection.free()**

- Memory is reclaimed earlier
- GC pass takes less time

# Queue

Removes the item least recently added – “first-in-first-out” (FIFO)



Minimum set of required functions (de.polygonal.ds.Queue<T> interface)

| | |
|---|---|
| enqueue() | Inserts a new element at the end of the queue |
| dequeue() | Removes and returns the element at the beginning of the queue |
| peek() | The element at the beginning of the queue (that has been present the longest) |

Applications

- Waiting lines, buffer for incoming data
- Simultaneous resource sharing by multiple consumers

More (based on *as3ds*)

↳ http://lab.polygonal.de/?p=189

# Queue Implementation

## de.polygonal.ds.ArrayedQueue<T>

- A circular array – the end of array “wraps around” to the start of the array
- Uses a fill count to distinguish between empty and full queues
- Insertion/removal of elements in constant time
- Best for fixed-sized queues → resizing a circular array is expensive
- Use dispose() to nullify last dequeued element to allow early garbage collection

## de.polygonal.ds.LinkedQueue<T>

- Implemented as a singly-linked list
- Fast peek() operation, but slower insertion/removal
- Best for queues of varying size and when maximum size is not known in advance
- More efficient than using the DLL<T> class for queue-like access

# Queue Example

## Example – using a queue to buffer up to x incoming elements

```
import de.polygonal.ds.ArrayedQueue;
var que = new ArrayedQueue<MyElement>(x);

…

if (que.isFull()) {
    que.dequeue(); //make room by dropping the "oldest" element
}

q.enqueue(element); //insert incoming element

//process buffered elements, from oldest to newest
for (i in 0...que.size()) {
    var element = que.get(i); //get(0) equals peek()
}
```

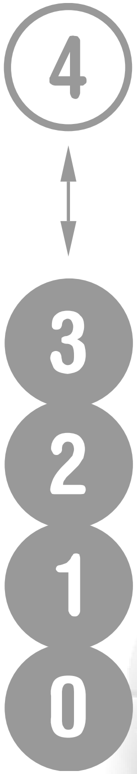
# Stack

Removes the item most recently added – “last-in-first-out” (LIFO)

All insertions and removals occur at one end (“top”) of the stack

Minimum set of required functions (de.polygonal.ds.Stack<T> interface)

| | |
|---|---|
| push() | Inserts a new element at the top of the stack |
| pop() | Removes and returns the element at the top of the stack |
| top() | The element at the top of the stack |



Applications – fundamental data structure

- Syntax parsing, expression evaluation, stack machines …
- Undo and backtracking

Common errors – stack underflow & overflow

- Underflow – pop (or peek at) an empty stack
- Overflow – push onto an already full stack

# Stack Implementation

## de.polygonal.ds.ArrayedStack<T>

- Insertion/removal just updates one variable (the stack pointer) → fast
- Use dispose() to nullify last popped off element to allow early garbage collection

## de.polygonal.ds.LinkedStack<T>

- Implemented as a singly-linked list
- Fast top() operation, but slower insertion and removal
- More efficient than using the SLL<T> class for stack-like access

## More stack methods for advanced usage

| | |
|---|---|
| dup() | Pops the top element of the stack, and pushes it back twice |
| exchange() | Swaps the two topmost elements on the stack |
| rotLeft(), rotRight() | Moves topmost elements in a rotating fashion |

# Stack Example

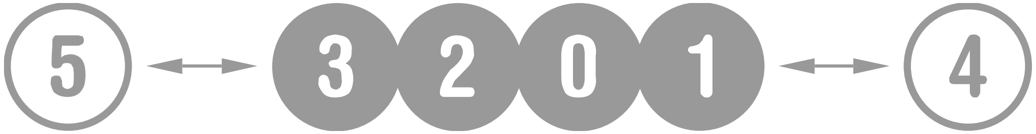
## Example – reversing data

```
var stack = new de.polygonal.ds.ArrayedStack<String>();

//push data onto stack
var input = "12345";
for (i in 0...input.length) {
    stack.push(input.charAt(i));
}

//remove data in reverse order
var output = "";
while (!stack.isEmpty()) {
    output += stack.pop();
}

trace(output); //outputs "54321"
```

# Deque

A deque is shorthand for “double-ended queue”

All insertions & deletions are made at both ends of the list



Minimum set of required functions (de.polygonal.ds.Deque<T> interface)

| | |
|---|---|
| pushFront() | Inserts a new element at the beginning (head) |
| popFront() | Removes the element at the beginning |
| pushBack() | Insert a new element at the end (tail) |
| popBack() | Removes the element at the end |

More

↳ http://lab.polygonal.de/?p=1472

# Deque Implementation

**de.polygonal.ds.ArrayedDeque<T>**

- Similar to STL deque implementation
  - Uses an array of smaller arrays
  - Additional arrays are allocated at the beginning or end as needed
- Amortized constant time complexity

**de.polygonal.ds.LinkedDeque<T>**

- Implemented as a doubly linked list
- More efficient than using the DLL<T> class for deque-like access

# Dense Array (DA)

Simulates a dense array by decorating a sparse array

Similar to flash.Vector.<T>

Fits nicely into existing Collection classes

Thanks to inlining performance on par with native array

Supports insertionsort → faster than quicksort for nearly sorted lists

Allows to move a block of data with memmove()

↳ http://www.cplusplus.com/reference/clibrary/cstring/memmove/

Allows removal of elements while iterating over it

# Dense Array (cont.)

Existing array method names are confusing …

- shift(), unshift() – feels like using unadd() for removing elements
- slice(), splice() – I always mix up both methods!

*ds* uses a different API

| | |
|---|---|
| pushBack(x:T):Void | Appends element |
| popBack():T | Removes last element |
| pushFront(x:T):Void | Prepends element |
| popFront():T | Removes first element |

| | |
|---|---|
| insertAt(i:Int, x:T):Void | Equals array.splice(i, 0, x) |
| removeAt(i:Int):T | Equals array.splice(i, 1) |
| removeRange(i:Int, n:Int):DA<T> | Equals array.slice(i, i + n – 1) |

# Dense Array Examples

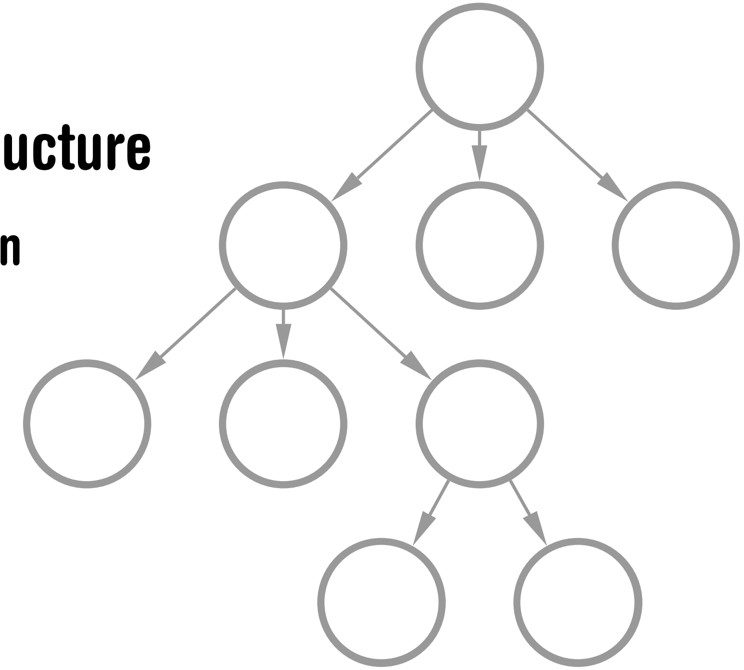
## Example – removal of elements in constant time if order doesn't matter

```
var denseArray = new de.polygonal.ds.DA<Int>();
for (i in 0...10) denseArray.pushBack(i);

//remove element at index 5
denseArray.swapWithBack(5);
denseArray.popBack();
```

## Example – resorting a nearly sorted array with insertion sort

```
import de.polygonal.ds.Compare;
import de.polygonal.ds.ArrayConvert;

var denseArray = ArrayConvert.toDA([0, 5, 1, 2, 3, 4]);
var useInsertionSort = true;
denseArray.sort(Compare.compareNumberRise, useInsertionSort);
```

# Tree



A tree is an acyclic graph (no cyclic paths)

Implemented as a hierarchical node-based structure

- Each node can store an arbitrary number of children
- Each node points to its parent and its first child
- Each node points to its next and previous sibling

Applications

- Representing hierarchical data like XML
- Scene graphs, bounding volume hierarchies (BVHs)
- Decision trees, story lines, component-based game architectures

More (based on *as3ds*)

↳ http://lab.polygonal.de/?p=184

# Tree Implementation

A node is represented by de.polygonal.ds.TreeNode<T>

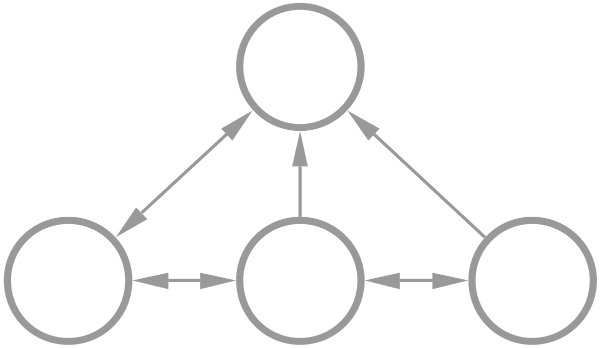
A tree is held together by linked nodes – there is no “tree manager” class

Class de.polygonal.ds.TreeBuilder<T> simplifies tree construction

A node contains …

| | |
|---|---|
| The node's data | TreeNode.val |
| The node's parent | TreeNode.parent |
| Reference to the first child | TreeNode.children |
| Reference to the next & previous sibling | TreeNode.left, TreeNode.right |

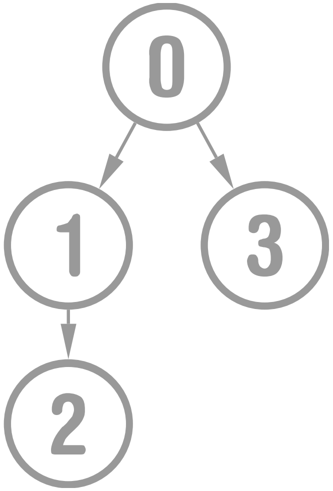
TreeNode pointers:



# Tree Construction Example

## Example – building a simple tree, top-down

```
import de.polygonal.ds.TreeNode;
import de.polygonal.ds.TreeBuilder;

var root = new TreeNode<Int>(0);

var builder = new TreeBuilder<Int>(root);
builder.appendChild(1);
builder.down();
builder.appendChild(2);
builder.up();
builder.appendChild(3);

trace(root); //outputs:
{TreeNode (root), children: 2, depth: 0, value: 0}
+---{TreeNode (child), children: 1, depth: 1, value: 1}
|   +---{TreeNode (leaf|child), depth: 2, value: 2}
+---{TreeNode (leaf|child), depth: 1, value: 3}
```



# Tree Traversal

## A traversal performs an action on each node (“visiting” a node)

- A traversal is initiated at the current node by calling one of the traversal methods
- The traversal then calls a function for each node in the subtree

## Example

```
node.preorder(…); //visit node and all descendants of node
```

## Depth-first traversal style

- Go deeper into the tree before exploring siblings
  - Preorder – visit root, traverse left subtree, traverse right subtree
  - Postorder – traverse left subtree, traverse right subtree, visit root

## Breadth-first traversal style

- Explore the breadth (“full width”) at a given level before going deeper
  - Levelorder – visit all nodes on each level together in order

# Tree Traversal (cont.)

**Elements can be visited by calling element.visit() …**

- All elements have to implement de.polygonal.ds.Visitable
- No anonymous function calls → fast

```
interface de.polygonal.ds.Visitable {
    function visit(preflight:Bool, userData:Dynamic):Bool;
}
```

**… or by passing a function reference to the traversal method**

```
function(node:TreeNode<T>, preflight:Bool, userData:Dynamic):Bool {…}
```

**In either case a traversal can be aborted by returning false**

**Parameters**

- “preflight” – if user returns false while preflight is true:
  - current node and all descendants are excluded from the traversal
- “userData” – stores custom data that gets passed to every visited node

# Tree Traversal Example 1

## Example – traversing a tree by using the Visitable interface

## Prerequisite

```
class Item implements de.polygonal.ds.Visitable {
    public var id:Int;
    public function new(id:Int) { this.id = id; }
    public function visit(preflight:Bool, userData:Dynamic):Bool {
        userData.push(id);
        return true;
    }
}
```

## Usage

```
var tree = … //see "Tree Construction Example” slide
var tmp = new Array<Int>; //stores node ids during traversal
var preflight = false;
var iterative = false;
tree.preorder(null, preflight, iterative, tmp);
trace(tmp.join()); //outputs "0,1,2,3”
```

# Tree Traversal Example 2

## Example – traversing a tree by passing a reference to a function

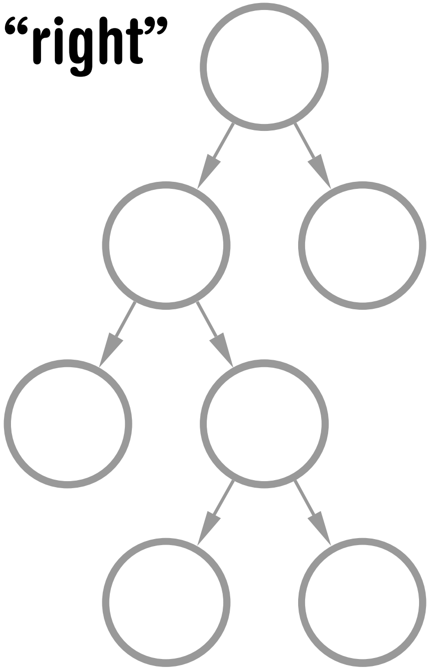
### Prerequisite

```
import de.polygonal.ds.TreeNode;
var visitor = function(node:TreeNode<Item>, userData:Dynamic):Bool {
    userData.push(node.val.id); //node.val points to Item
    return true;
}
```

### Usage

```
var tree = … //see "Tree Construction Example” slide
var tmp = new Array<Int>; //stores node ids during traversal

var iterative = false;
var list = new Array<Int>;
tree.postorder(visitor, iterative, tmp)
trace(list.join()); //outputs "2,1,3,0”
```

# Binary Tree



A subset of a tree – at most two children called the “left” and “right”

Can be implicitly stored as an array

- Wastes no space for complete binary trees (see Heap)

Supports inorder traversal – visit left child, root, right child

de.polygonal.ds.BinaryTreeNode<T>

- A node-based binary tree similar to the TreeNode class

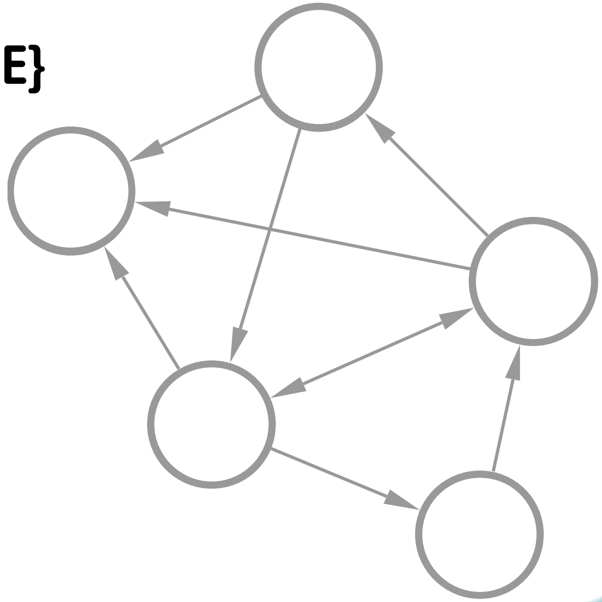
Applications

- Many advanced tree-based structure use a binary tree as its base
  - Heaps, self-balancing binary search trees (e.g. AVL, Red-black tree)
- Binary Space Partition (BSP) trees
  - Visibility determination, spatial data partitioning

# Graph

A graph is a symbolic representation of a network of any kind

- Formal: A set of nodes N, linking with the set of edges, E: G = {N, E}
- Nodes are called "vertices", edges are also called "arcs"



*ds* includes a uni-directional weighted graph

- Uni-directional → an edge is a one-way connection
- Weighted → an edge has a cost to go from one node to the next

Implemented as an adjacency list

- Efficient storage of sparse graphs (few connections per node)
- Sparse graphs are more common than dense graphs

Graph theory is complex and has many applications

More (based on *as3ds*)

↳ http://lab.polygonal.de/?p=185

# Graph Implementation

## de.polygonal.ds.Graph<T>

- Manages graph nodes
- Provides methods for adding/removing graph nodes
- Provides methods for searching the graph

## de.polygonal.ds.GraphNode<T>

- Stores the node's data
- Stores additional information while running a graph search algorithm
- Stores arcs (connections) to other nodes

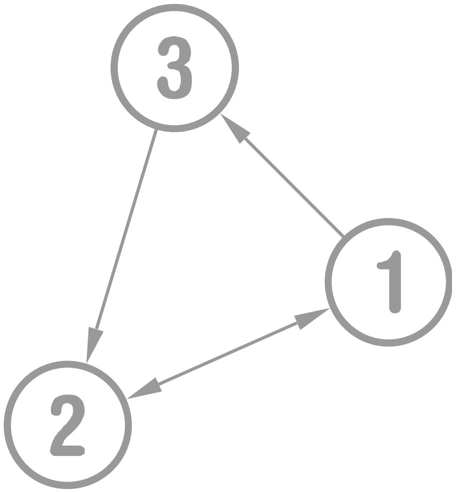
## de.polygonal.ds.GraphArc<T>

- A connection to another node
- Stores the node that the arc is pointing at

# Graph Construction Example

## Example – building a simple graph

```
import de.polygonal.ds.Graph;
import de.polygonal.ds.GraphNode;

var graph = new Graph<Int>();
var node1:GraphNode<Int> = graph.addNode(1);
var node2:GraphNode<Int> = graph.addNode(2);
graph.addMutualArc(node1, node2);

var node3:GraphNode<Int> = graph.addNode(3);
graph.addSingleArc(node1, node3);
graph.addSingleArc(node3, node2);
```



# Graph Search Algorithms

Depth-first search (DFS) – “long and stringy”

- Start at initial node (“seed”) and follow a branch as far as possible, then backtrack
- Closely related to preorder traversal of a tree

Breadth-first search (BFS) – “short and bushy”

- Start at root node (“seed”) and explore all the neighboring nodes first

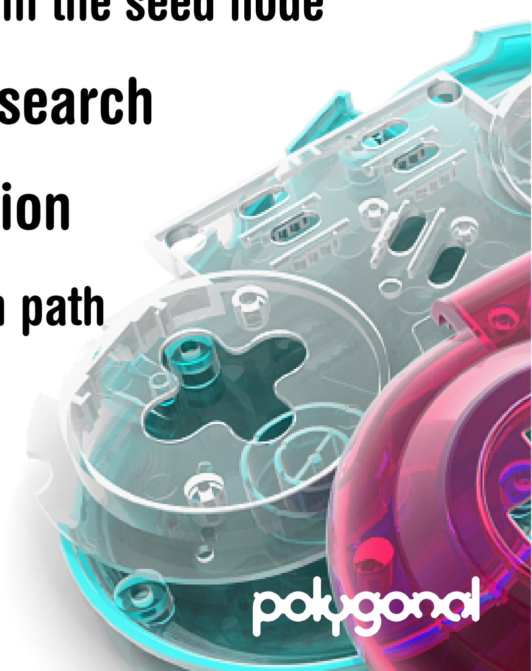
Depth-limited breadth-first search (DLBFS)

- Same as BFS, but only explore neighbors within a maximum distance from the seed node

Call Graph.clearMarks() to make the entire graph visible to the search

After running a BFS/DFS, GraphNode stores additional information

| | |
|---|---|
| GraphNode.parent | the previously visited node to backtrack the search path |
| GraphNode.depth | the traversal depth (distance from seed node) |



# Graph Search Example

## Example – searching a graph (very similar to tree traversal)

```
import de.polygonal.ds.Graph;
import de.polygonal.ds.GraphNode;

var graph = new Graph<Int>();
…
var f = function(node:GraphNode<T>, preflight:Bool, userData:Dynamic):Bool {
    trace("searching: " + node.val);
    return true;
}

var preflight = false;
var seed = graph.nodeList; //use first node as initial node
graph.DFS(preflight, seed, f);
```

## More

↳ **http://lab.polygonal.de/?p=1815**

# Heap

A heap is a special kind of binary tree satisfying the “heap property”

- Every parent element is greater than or equal to all of its children
- Satisfy denseness → nodes are packed to the left side in the bottom level

Insertions & deletions are done in logarithmic time

Called a min-heap if smallest element is stored in the root (otherwise a max-heap)

Minimum set of required functions

- Insert element
- Return/delete the smallest (or largest) element

All elements have to implement Comparable<T> interface

```
interface de.polygonal.ds.Comparable<T> {
    function compare(other:T):Int;
}
```

# Heap (cont.)

## A heap can be transformed into a sorting algorithm called Heapsort

- Heap is build by inserting elements, then removing them one at a time – elements come out in order from smallest to largest or v.v.
- In-place and with no quadratic worst-case scenarios → see Heap.sort()

## Elements are partially-sorted!

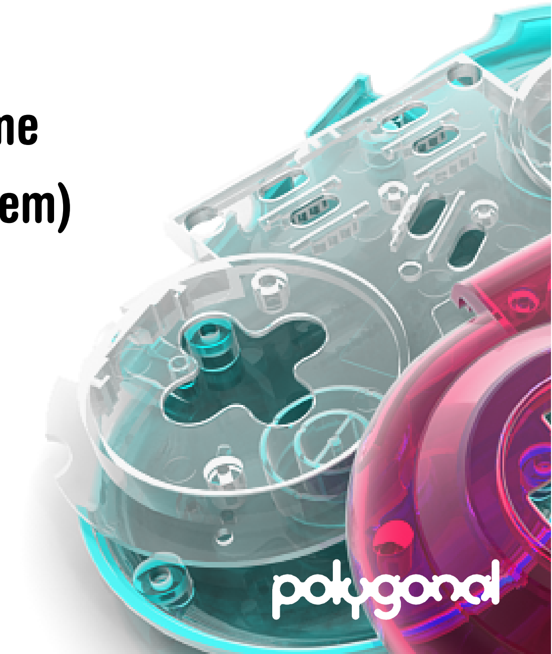
| | |
|---|---|
| Heap.iterator() | returns all elements in random order since performance matters |
| Heap.toString() | returns elements in sorted order |

## Applications

- Finding the min, max, k-th largest element in linear or even constant time
- Graph algorithms (minimal spanning tree, Dijkstra's shortest path problem)
- Job scheduling (element key equals event time)

## More

↳ http://lab.polygonal.de/?p=1710

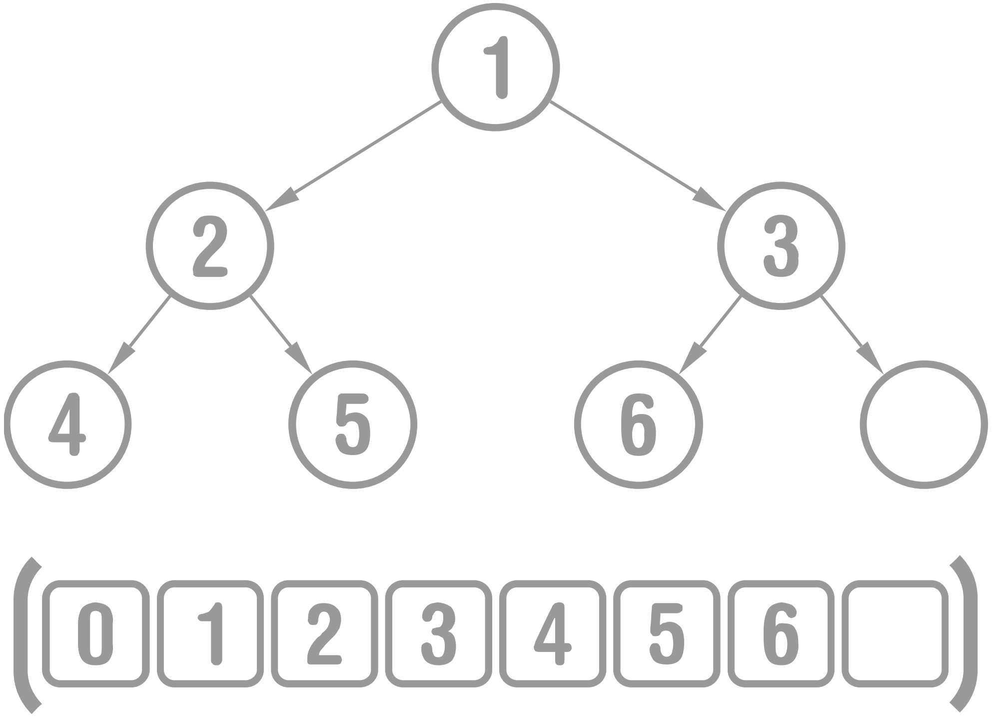


# Heap Implementation

The heap structure is a binary heap and implicitly stored as an array

- Every item i has a parent at index i/2, a left child at index i*2 and a right child at i*2+1
- The tree needs to be dense
- No linked implementation as inefficient in most cases

Array locations:



# Heap Example

## Example – finding the minimum value in a heap of random integers

## Prerequisite

```
class HeapItem implements de.polygonal.ds.Heapable<HeapItem> {
    public var value:Int;
    public var position:Int; //internal use, never change!
    public function new(value:Int) {
        this.value = value;
    }
    public function compare(other:HeapItem) {
        return other.value - value; //sort smallest to largest
    }
}
```

## Usage

```
var heap = new de.polygonal.ds.Heap<HeapItem>();
for (i in 0...10)
    heap.add(new HeapItem(cast Math.random() * 100);
trace(heap.pop()); //outputs minimum value
```

# Priority Queue

A queue that keeps its elements sorted in order of priority

Elements are records with numerical keys representing priority values

Implementation uses an optimized & simplified Heap class

Priority values are constrained to floats

- Fast inlined float comparison instead of custom comparison function

Minimum set of required functions

- Insert element with an assigned priority
- Return and remove element with highest priority

All elements have to implement Prioritizable interface

```
interface de.polygonal.ds.Prioritizable {
    var priority:Float;
    var position:Int;
}
```

# Priority Queue Example

## Prerequisite

```
class PrioritizedItem implements de.polygonal.ds.Prioritizable {
    public var priority:Int;
    public var position:Int; //internal use, never change!

    public function new(priority:Float) {
        this.priority = priority;
    }
}
```

## Usage

```
import de.polygonal.ds.PriorityQueue;

//by default a higher number equals a higher priority
var pq = PriorityQueue<PrioritizedItem>();

pq.add(new PrioritizedItem(5));
pq.add(new PrioritizedItem(1));
trace(heap.pop()); //outputs element with priority 5
```

# Map

Also known as “associative array” or “dictionary”

Abstract data type – concrete maps implement de.polygonal.ds.Map<K,T>

A collection of (key, value) pairs

- Keys are unique
- Each key maps a single value

Minimum set of required functions

- Add and remove (key, value) pairs
- Modify values of existing pairs
- Find the value for a key

Implementations

| | |
|---|---|
| HashMap<K,T> | A simple wrapper for the Flash Dictionary class |
| *HashTable classes | Arrayed hash tables (cross-platform) |

# Hash Table

Implemented as arrayed hash tables

Storage and retrieval of data in constant time on average

Uses a hash function for mapping keys to values

- Transforms a key to an index ("hash") into an array element ("bucket")

Does not depend on Flash Dictionary class, yet high performance

Gives full control over memory usage v performance

Flavors

| | |
|---|---|
| • IntIntHashTable | Maps integer keys to integer keys |
| • IntHashTable<T> | Maps integer keys to objects |
| • HashTable<K, T> | Maps keys of type Hashable to objects |

More

↳ http://lab.polygonal.de/?p=1325

# Hash Table – Keys

Keys used in a HashTable<K,T> have to implement de.polygonal.ds.Hashable or just extend from de.polygonal.ds.HashableItem

```
interface de.polygonal.ds.Hashable {
    var key:Int; //internal use, never change!
}
```

KISS solution of platforms which can't use objects as keys

## Example

```
import de.polygonal.ds.HashKey;
class CustomKey implements de.polygonal.ds.Hashable {
    public var key:Int;
    //assign unique key by incrementing a counter in HashKey
    public function new() { key = HashKey.next(); }
}
```

## Usage

```
myHashTable.set(new CustomKey(), myValue);
```

# Hash Table – Multiple Values/Key

set() allows to map multiple values to the same key

Values are managed in a “first-in-first-out” manner

Example

```
import de.polygonal.ds.IntIntHashTable;
var hash = new IntIntHashTable();

var addedFirstTime:Bool = hash.set(1, 5); //true
var addedFirstTime:Bool = hash.set(1, 6); //false: key 1 now maps values 5,6

var value = hash.get(1); //equals 5
hash.remove(1);
var value = hash.get(1); //equals 6
```

Strict map behavior can be enforced with helper method setIfAbsent() or by using has() before set()

# Set

Stores unique values without any particular order

Abstract data type – concrete sets implement de.polygonal.ds.Set<T>

Uses same implementation as in hash tables classes

Implementations

| | |
|---|---|
| IntHashSet | Stores integer values |
| HashSet<T> | Stores objects using an arrayed hash table |
| ListSet<T> | Stores objects using an array – simple & efficient for small sets |

# Bit Vector

An array of bits (also called “bit array”)

Packs 32 boolean values into a 32-bit integer (4 bytes)

Efficient → an array of boolean types would require 128 bytes (32 * 4)

Fixed capacity → needs to be resized manually

de.polygonal.ds.BitMemory – a fast bit vector using alchemy memory

# The Collection Interface

# Collection

A collection is an object that stores other objects (its elements)

All structures implement de.polygonal.ds.Collection<T>

Interface methods

```
function free():Void
function contains(x:T):Bool
function remove(x:T):Bool
function clear(purge:Bool = false):Void
function iterator():Itr<T>
function isEmpty():Bool
function size():Int
function toArray():Array<T>
function clone(assign:Bool = true, ?copier:T->T):Collection<T>
```

# Collection.free()

Destroys an object by nullifying its internals

Ensures objects are GCed as early as possible

- Lower memory usage
- Less noticeable lags from running the GC

Mandatory for data structures using “virtual memory” to prevent a memory leak

- Used in all *HashTable and *HashSet classes
- Used in de.polygonal.ds.mem.*

Recommended for complex, nested and linked structures

# Collection.free() – Example

## Example – tearing a linked list apart

```
class Foo {
    public var node:de.polygonal.ds.DLLNode<Foo>;
    public function new() {}
}

…
onEnterFrame = function() {
    var list = new de.polygonal.ds.DLL();
    for (i in 0...100000) { //create tons of objects per frame
        var foo = new Foo();
        foo.node = list.append(foo); //circular reference
    }
    list.free();
}
```

## Benchmark results (FlashPlayer 10.1.85.3, Windows)

- **Average memory usage drops from 56 megabytes to 7 megabytes**
- **Average frame rate increases from 23fps to 29fps**

# Collection.clear()

**Method signature**

```
function clear(purge:Bool = false):Void
```

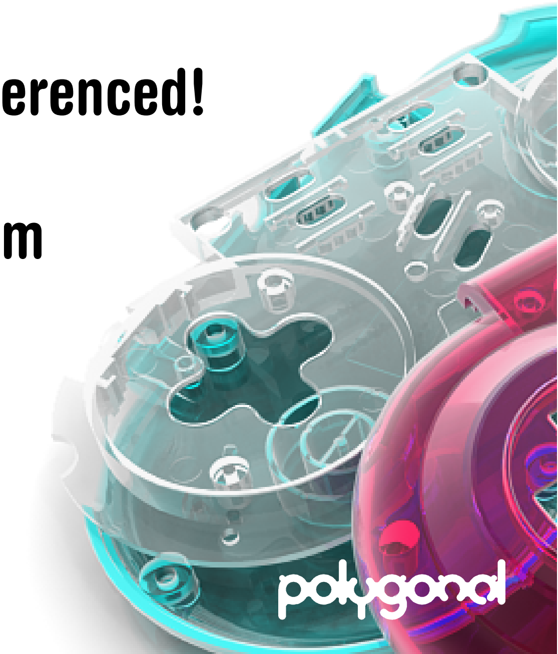
**Removes all elements from a collection**

***ds*** **does nothing to ensure empty array locations contain null**

**For example, a stack just sets the stack pointer to zero**

**This is fast but objects can't be GCed because they are still referenced!**

**Call clear(true) to remove elements and to explicitly nullify them**



# Collection.iterator()

**Process every element without exposing its underlying implementation**

**No specific order → see documentation for implementation details**

**Example – explicit iterator**

```
var iterator:Iterator<T> = myCollection.iterator();
while (itr.hasNext()) {
    var item:T = itr.next();
    trace(item);
}
```

**Example – implicit iterator in haXe**

```
for (item in myCollection) trace(item);
```

- **No boilerplate code**
- **Works with all objects that are Iterable (have an iterator() method)**

# Collection.iterator() – Lambda

**An interface between collections and algorithms**

**Allows generic algorithms to operate on different kinds of collections**

**Example**

```
import de.polygonal.ds.Collection;
import de.polygonal.ds.ArrayConvert;

var collection:Collection<Int> = ArrayConvert.toDLL([1, 2, 3]);

var exists = Lambda.exists(dll, function(x) return x == 2);
trace(exists); //true

var result = Lambda.fold(dll, function(a, b) return a + b, 0);
trace(result); //6 (1+2+3)
```

**More**

↳ **http://haxe.org/api/lambda**

# Collection.iterator() – Itr.reset()

## Interface Itr<T> defines an additional reset() method

- Avoid frequent allocation by reusing existing iterator object multiple times

## Example

```
var iterator:de.polygonal.ds.Itr<T> = myCollection.iterator();
for (i in iterator) process(i);
iterator.reset();
for (i in iterator) process(i);
```

## Many collections have a boolean field named reuseIterator

- If set to true, a single internal iterator object is allocated and reused
- Less verbose than calling reset() – but use with care:

```
myCollection.reuseIterator = true;
var iteratorA:Itr<T> = myCollection.iterator();
var iteratorB:Itr<T> = myCollection.iterator();
trace(iteratorA == iteratorB); //true
```

# Collection.iterator() – Itr.remove()

## Interface Itr<T> defines an additional remove() method

- Allows removal of elements while iterating over a collection
- Convenient since no marking or temporal storage is required

## Example

```
var myCollection = …

var iterator:Itr<T> = myCollection.iterator();
while (itr.hasNext()) {
    var item = itr.next();
    itr.remove(item);
}
```

## Never modify a collection during an iteration!

- Itr.remove() is the only safe operation

# Collection.iterator() – Performance

## Don't use iterators for performance-critical code

- Overhead from calling hasNext() and next() for every item
- Sacrifice syntax sugar and use “raw” loops instead

## Example – traversing a linked list

```
var node = myDoublyLinkedList.head;
while (node != null) {
    var item = node.val;
    node = node.next;
}
```

## Example – traversing an arrayed queue

```
for (i in 0...que.size()) {
    var item = que.get(i);
}
```

- Close to native performance – size() is evaluated once, get() is inlined

# Collection.clone()

Method signature

```
function clone(assign:Bool = true, ?copier:T->T):Collection<T>
```

There are two ways to clone a data structure – „shallow“ and „deep“

- Shallow mode – copies the structure by value and its elements by reference (default)
- Deep mode – copies the structure and its elements by value

There are two ways to create deep copies

- All elements implement Cloneable<T> interface

```
myCollection.clone(false);
```

- User passes a function responsible for cloning elements

```
myCollection.clone(false, func);
```

User choice!

# Collection.clone() – Example

## Prerequisite

```
class Foo implements de.polygonal.ds.Cloneable<Foo> {
    public var value:Int;
    public function clone():Foo { return new Foo(value); }
}
...
var myList = new de.polygonal.ds.SLL<Foo>();
```

## Example – shallow copy

```
var copy:SLL<Foo> = cast myList.clone();
```

## Example – deep copy using cloneable interface

```
var assign = false;
var copy:SLL<Foo> = cast myList.clone(assign);
```

## Example – deep copy using a function

```
function cloneFunc(source:Foo) { return new Foo(source.val); });
var copy:SLL<Foo> = cast myList.clone(assign, cloneFunc);
```

# Thanks for your attention!