Abstract Data Types

2501ICT/7421ICTNathan

René Hexel

School of Information and Communication Technology
Griffith University

Semester 1, 2012

Outline

Collections

- ADTs and Data Structures
- Collection Categories
- Common Collection Operations
- Traversal
- Serialisation
- Collection Implementations
- C, Objective-C

ADTs

- Abstract Data Type
- Describes a collection of data items and the associated operations that can be applied
- High-level concept of data organisation (what)
- Data Structure
- Physical Implementation of an ADT
- How to represent ADT concept
- There is no ‘best’ Implementation under All Conditions

Collections

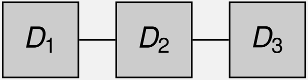
- Definition
  - a Collection is a *group of items* forming a conceptual unit
- Collections are
  - $\rightarrow$ represented by ADTs, and
  - $\rightarrow$ implemented through Data Structures

Collection Categories

- Linear
  - Arrays, Lists, Stacks, Queues, ...
- Hierarchical
  - Heaps, Trees, Hashes, ...
- Connected
  - Graphs
- Unordered
  - Sets, Bags, Maps, ...

$\rightarrow$ Play an Important role in almost all Non-Trivial Programs!

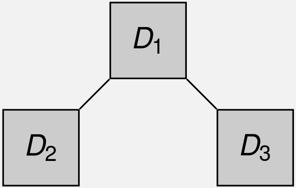
Linear Collections



- Low Level Properties
  - Array/List operations
  - e.g. dynamic array, singly/doubly linked list
- High Level View
  - Stack/Queue/…functionality
  - e.g. Push-Down Stack, Priority Queue, Pipe

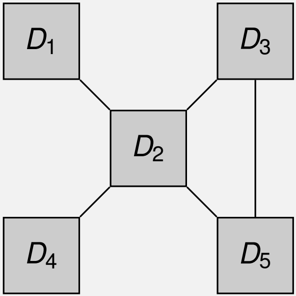
Hierarchical Collections

- Binary Tree
- Binary Search Tree
- Generic Tree
- Heap
- Red/Black Tree
- ...



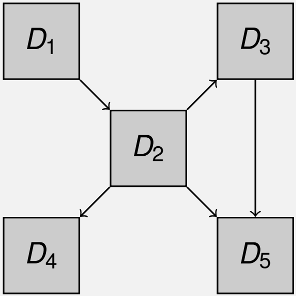
Graphs

- Undirected Graph
  - Nondirectional



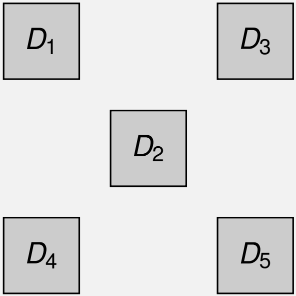
Graphs (2)

- Directed Graph
  - Directional



Unordered Collections

- Set
- Bag
- Map
  - Dictionary
  - Table



Common Operations

- Search and Retrieval
  - → given certain search criteria (search properties)
  - → return item or position (if found)
  - → return distinguishing value like `nil` or -1 if not found
- Removal
  - → delete a given item
  - → delete item at specific position

Common Operations (2)

- Insertion
    - add a new item
    - usually at some particular position
    - → e.g. at head, at tail, after item x, . . .
- Replacement
    - combination of Removal and Insertion
    - ‘in place’ replacement
    - → when atomic action is required

Common Operations (3)

- Traversal
  - visit each item
  - “do something” with that item
- Test (the whole collection)
  - for equality
  - greater than, more elements
  - less than, fewer elements

Common Operations (4)

- Size
  - Number of items
  - Byte size
- Cloning
  - ‘deep copy’
  - copy an entire collection
  - each item needs to be cloneable!

Traversal

- Need for Sequential Traversal
- ADTs differ in Data Organisation
- Enumerator (iterator) is required
- Object or Function that makes traversal possible
- `-nextObject` for `NSEnumerator`
  - return next item and advance
  - return `nil` if no next item exists
- `++` for STL `iterators`
- `begin()` and`end()` methods mark start and beginning

Casting

- Problem: Collections may Contain Objects of Any Type
- Cast to required subclass necessary in C++ for non-virtual methods!
  - → unchecked: `(string)` – also works in C, Objective-C
  - → checked: `static_cast<string>` – C++ only
- Primitive types (e.g. int) need a wrapper
  - → e.g. `NSNumber` in Objective-C
  - → e.g. `intValue` and `setIntValue:` access methods
- limited compile-time type checking!
- type of object can be tested during run-time using `isKindOf:`

Casting in plain C

- No ‘Collection’ Infrastructure
- Implement your own or use add-on library
- Use `void *` for generic objects
- Cast to Required ‘Object’ Pointer
- Primitive types (e.g. `int`) may need a wrapper
- No compile-time type checking – type of object cannot be tested during run-time!
- Casting to wrong pointers causes crashes!

Saving Data on Disk

- Serialised stream representation of each object
- `NSCoding` Objective-C Protocol for write/read
  - `- encodeWithCoder:` and `- initWithCoder:`
  - Abstract set of Methods (`NSCoder`)
  - → Structured Files and XML: `NSKeyedArchiver`, `NSKeyedUnArchiver`
  - → Network Connections: `NSPortCoder`
- No API support in C++
  - → traverse STL collection using `iterator`
  - → read/write data using `stream` classes

Saving Data in Objective-C

Example

```
#import <Foundation/Foundation.h>

int main(int argc, char *argv[])
{
    @autoreleasepool
    {
        NSArray *array = [NSArray arrayWithObjects: @"1", @"2", @"3", @"4", nil];

        /*
         * save the array to disk, to a file called "Array.bin"
         */
        [NSKeyedArchiver archiveRootObject: array toFile: @"Array.bin"];
    }

    return EXIT_SUCCESS;
}
```

Loading Data in Objective-C

Example (prints: `Array:  1 2 3 4 `)

```
#import <Foundation/Foundation.h>

int main(int argc, char *argv[])
{
    @autoreleasepool
    {
        /*
         * load the array from disk (from the "Array.bin" file)
         */
        NSArray *array = [NSKeyedUnarchiver unarchiveObjectWithFile: @"Array.bin"];

        NSEnumerator *e = [array objectEnumerator];
        id object;
        printf("Array:  ");
        while ((object = [e nextObject]) != nil)
                printf("%s ", [[object description] UTF8String]);
        putchar('\n');
    }

    return EXIT_SUCCESS;
}
```

Saving Data in C++

Example

```
#include <iostream>
#include <fstream>
#include <vector>
#include <string>

using namespace std;

int main(int argc, char *argv[])
{
        vector<int> array(5, 0);                         // an array of ints
        for (int i = 0; i < 5; i++) array[i] = 2*i;     // initialise

        /*
         * write vector to a file "Vector.txt"
         */
        fstream outputFile("Vector.txt", fstream::out);
        vector<int>::iterator enumerator = array.begin();
        while (enumerator != array.end())
                outputFile << *enumerator++ << endl;
        outputFile.close();

        return outputFile.fail() ?  EXIT_FAILURE : EXIT_SUCCESS;
}
```

Loading Data in C++

Example (prints: `0 2 4 6 8 `)

```
#include <iostream>
#include <fstream>
#include <vector>
#include <string>

using namespace std;

int main(int argc, char *argv[])
{
        vector<int> array(0);                              // an array of ints
        string line;

        fstream inputFile("Vector.txt", fstream::in);    // read vector from file
        while (getline(inputFile,line))
        {
                int element = atoi(line.c_str());        // convert line to int
                array.push_back(element);                // add to array
        }
        vector<int>::iterator enumerator = array.begin();
        while (enumerator != array.end())                // traverse array
                cout << *enumerator++ << " ";
        cout << endl;

        return !inputFile.eof() ?  EXIT_FAILURE : EXIT_SUCCESS;
}
```

Machine-Independent Files

- Use Text-Based File Format
  - → e.g. XML
- Write: Traverse each Element
  - print out content into stream, e.g. using `fprintf()`, `-encodeWithCoder:`, ...
- Read: Parse input Data ⇒ e.g. `NSScanner`
  - → create Data Structure as you go
  - → add each one to your collection

Saving XML Data in Objective-C

Example

```
#import <Foundation/Foundation.h>

int main(int argc, char *argv[])
{
    @autoreleasepool
    {
        NSArray *array = [NSArray arrayWithObjects: @"1", @"2", @"3", @"4", nil];
        NSMutableData *data = [NSMutableData new];
        NSKeyedArchiver *archiver = [[NSKeyedArchiver alloc]
                                initForWritingWithMutableData: data];
        [archiver setOutputFormat: NSPropertyListXMLFormat_v1_0];
        [archiver encodeObject: array forKey: @"root"];        // create XML
        [archiver finishEncoding];
        [archiver release];

        [data writeToFile: @"Array.xml" atomically: YES];      // write file
        [data release];
    }

    return EXIT_SUCCESS;
}
```

Loading XML Data in Objective-C

Example (prints: `Array:  1 2 3 4`)

```
#import <Foundation/Foundation.h>

int main(int argc, char *argv[])
{
    @autoreleasepool
    {
        /*
         * load the array from disk (from the "Array.xml" file)
         */
        NSArray *array = [NSKeyedUnarchiver unarchiveObjectWithFile: @"Array.xml"];

        NSEnumerator *e = [array objectEnumerator];
        id object;
        printf("Array:  ");
        while ((object = [e nextObject]) != nil)
                printf("%s ", [[object description] UTF8String]);
        putchar('\n');
    }

    return EXIT_SUCCESS;
}
```

ADT Implementations

- High-level Decisions
  - accessing head/tail only?
  - random access needed?
  - map/dictionary functionality needed?
- Low-level Decisions
  - arrays (static vs. dynamic)
  - linked lists (linked data structures)
  - hash maps

ADT Implementations (2)

- Often: Multiple Implementations
  - time/space tradeoff
- Linear Collections
  - arrays
  - linked lists
  - hash maps
- Hierarchical Collections
  - different linkage models

Arrays

- One of the Most Commonly Used Low Level Data Structures
- Access Elements by Index Position
- Index Operation is Very Fast
- Constant time to access any element
- Element position does not affect access speed

Physical Array Size

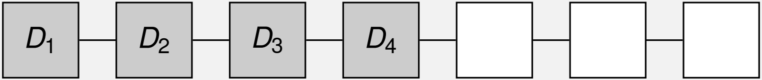
- Capacity (max. size) of an Array
- C: use sizeof (size in Bytes!)

  ```
  char *students[100];
  sizeof(students) / sizeof(students[0]) = 100;
  ```
- Objective-C
  - transparent, but `NSMutableArray` can be optimised for a given capacity!

    ```
    NSMutableArray s = [NSMutableArray
                         arrayWithCapacity: 100];
    ```
- C++ `vector` class is also transparent, but optimised for a fixed size

Logical Array Size

- Number of Valid Items
- e.g. 4 items have been added to the array:



- A Dedicated Counter Variable is needed to Keep Track of the Size

Adding an Item to an Array

- Check if logical size equals physical size
- If so, we need to Increase the size of the array:
- Create a new, larger array
- Copy old array to new array
- Refer the old array variable to the new array
- Add the Item to the new array

Resizing in Plain C

Example (expanding an array in C)

```
struct Object *temp;
int i;

if (logsz == physz)
{
  physz++;                                   /* increase physical size */
  temp = malloc(physz * sizeof(struct Object)); /* allocate new memory */

  for (i=0; i < physz; i++)                  /* copy old array to new */
    temp[i] = array[i];

  free(array);                               /* free old array */
  array = temp;                              /* use new array */
}
```

Complexity

- Resizing is Costly
  - complexity shoots up from O(1) to O(n)

  $\Rightarrow$ resize less often
- Don't just add 1, but double the size each time:

```
physz *= 2;
```

Decreasing Size

- Similar to Increasing
- Frees up wasted space

$\rightarrow$ Create a temporary, smaller array
  - As costly as increasing

  $\Rightarrow$ Don't decrease too often!
- Good strategy: decrease only if the logical size is less than $\frac{1}{4}$ of the physical size

  $\rightarrow$ Decrease only by $\frac{1}{2}$ to leave room for adding elements again

Inserting an Item

- Check Available Space
- Resize if necessary: O(n)
- Shift Items from target index to logical end one index down: O(n)

D_1	D_2	D_3	
D_1	D_2		D_3
D_1	D_2	D_4	D_3

Removing an Item

- Shift Items from target index + 1 to logical end one index up: O(n)
- Check Wasted Space
- Decrease size if necessary: O(n)

D_1	D_2	D_3	D_4
D_1	D_2		D_4
D_1	D_2	D_4	

Array Problems

- Insertions and Deletions incur some overhead
- Shifting items to open or close a gap
- Copying all items when resizing
- O(n) Complexity in Time and Space
- Only efficient if mostly static!

Array Problems (2)

- Require Contiguous Memory
- Expensive for large data structures
- 1:1 Correspondence Between
  - logical position of a cell and its
  - physical position in memory
- Decouple Logical/Physical Pos.

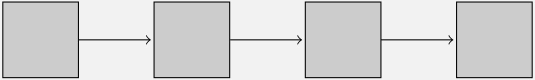
  $\rightarrow$ linked data structure

Linked Data Structures

- Consist of Elements Called Nodes
- A Node Contains
  - The actual data
  - One or more links to other nodes
- Dynamic Data Structures
- Memory is allocated only as needed
- Can be freed immediately if unneeded

Singly Linked Lists

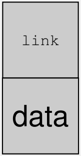
- Illustration:



- Accessing a Node (an Item)
- Follow the links from the Head
- Last item has a null link
- Dummy link indicating the end of the list

Nodes

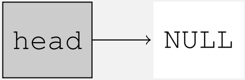
- Illustration of a Node



- A Node stores
  - a Pointer to another Node (`link`)
  - an Object (the actual data)

Singly Linked Structures

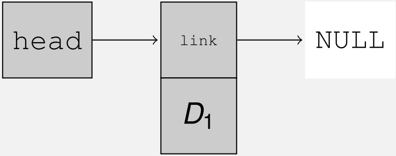
- Start With a Null Pointer (`nil` or `NULL`)



- Called the Head Pointer or an External Pointer
- Contains no data!

Singly Linked Structures (2)

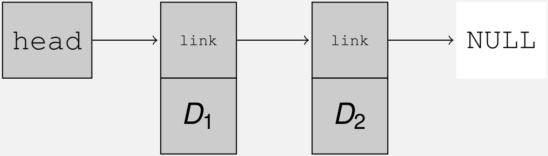
- Add the first Node



- Node Contains Actual Data
- Let `head` point to this first node
- Node itself points to `NULL` link as next node

Singly Linked Structures (3)

- Add the second Node

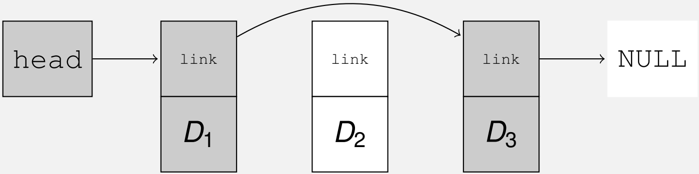


- Then add third, fourth, etc.
- All Nodes contain their own data and
  - → a `NULL` Pointer when added at the tail

→ Pointers get updated as new Nodes are added

Removing a Node

- To delete a Node



- Aim predecessor (or `head`) pointer at following node
- Release used up memory

Singly Linked List Complexity

- Adding an Item: $O(n)$
- Deleting an Item: $O(n)$
- Linear Search: $O(n)$
- Binary Search: $O(n \log n)$
- No direct access possible!
- Cache pointers
- Reduce frequent operations to $O(1)$