Proceedings of the 6th STL Workshop

CPH STL Report 2006-8, October 2006

Editor

Jyrki Katajainen

Contributors

Gerth Stølting Brodal
Hervé Brönnimann
Manuel Macías Córdoba
Miguel Fiandor Gutiérrez
Kasper Egdø
Amr Elmasry
Nicolai Esbensen
Claus Jensen
Jyrki Katajainen
Rune Møllegård Madsen
Pat Morin
Gabriel Moruz

Preface

This volume contains the papers presented at the 6th STL Workshop which was organized at the University of Copenhagen on October 30, 2006. The workshop was part of a course where the goal was to train students in scientific work processes. Of the seven students, who registered for this course, five wrote a paper that is included in this volume. The workshop program was enriched with presentations by Gerth Stølting Brodal (University of Aarhus), who also functioned as an external examiner for the students, and Claus Jensen (University of Copenhagen) and myself. This volume also includes the abstracts of these presentations.

In the call for papers, papers presenting original research in the area of program library development were sought. The topics included, but were not limited to:

- data structures and algorithms
- generic programming
- program library design
- programming language support
- software tools
- use of program libraries.

As you can read from this volume, the papers presented touch many of the areas mentioned above.

I thank Andrei Voronkov (University of Manchester) for letting us use the EasyChair conference management system and http://www.easychair.org for hosting the submission server. EasyChair was a useful tool in our role play as authors and program-committee members. I also thank the Danish Natural Science Research Council for their financial support (under contract and 272-05-0272; project "Generic programming—algorithms and tools"). Finally, I thank the students and the invited speakers for their contributions which made this workshop possible.

October 2006 Jyrki Katajainen

Sponsors

The Danish
Natural Science
Research Council

EASY CHAIR

Table of contents

Skewed binary search trees

Gerth Stølting Brodal Gabriel Moruz

Department of Computer Science, University of Aarhus, IT Parken, Åbogade 34, DK-8200 Århus N, Denmark
`gerth@daimi.au.dk` | `gabi@daimi.au.dk`

Invited speaker. Gerth Stølting Brodal

Abstract. We consider in this talk the class of balanced binary search trees, where the left subtree of a node v stores a fraction $\alpha \in [0, 1]$ of the nodes in the subtree rooted at v. Surprisingly, we show that skewed balanced search trees with $\alpha \approx 0.3$ outperform perfectly balanced search trees for uniformly distributed searches on a set of experiments involving 20.000 elements. This observation is explained by the fact that searching left and right of a node can have different costs because of various CPU features, including branch prediction schemes and cache replacement strategies. We show experiments for various layouts of skewed balanced trees. The most efficient layout in the experiments is a variation of the van Emde Boas layout adapted to skewed balanced search trees.

The full version of this paper has appeared in the *Proceedings of the 14th Annual European Symposium*, *Lecture Notes in Computer Science* **4168**, Springer-Verlag (2006), 708–719. Also available in electronic form at `http://www.daimi.au.dk/~gerth/Papers/esa06skew.pdf`.

The accompanying presentation is available at `http://www.daimi.au.dk/~gerth/Slides/cphstl06.pdf`.

Two-tier relaxed heaps*

Amr Elmasry[1] Claus Jensen[2] Jyrki Katajainen[2]

[1] *Department of Computer Engineering and Systems*
Alexandria University, Alexandria, Egypt
`aelmasry5464@yahoo.com`
[2] *Department of Computing, University of Copenhagen*
Universitetsparken 1, 2100 Copenhagen East, Denmark
`surf@diku.dk` | `jyrki@diku.dk`

Invited speaker. Claus Jensen

Abstract. We introduce an adaptation of run-relaxed heaps which provides efficient heap operations with respect to the number of element comparisons performed. Our data structure guarantees the worst-case cost of $O(1)$ for *find-min*, *insert*, and *decrease*; and the worst-case cost of $O(\lg n)$ with at most $\lg n + 3 \lg \lg n + O(1)$ element comparisons for *delete*, improving the bound of $3 \lg n + O(1)$ on the number of element comparisons known for run-relaxed heaps. Here, n denotes the number of elements stored prior to the operation in question, and $\lg n$ equals $\max\{1, \log_2 n\}$.

The full version of this paper will appear in the *Proceedings of the 17th International Symposium on Algorithms and Computation*, *Lecture Notes in Computer Science* **4288**, Springer-Verlag (2006), 308–317.

The accompanying presentation is available at the CPH STL home page `http://www.cphstl.dk`.

*Partially supported by the Danish Natural Science Research Council under contracts 21-02-0501 (project Practical data structures and algorithms) and 272-05-0272 (project Generic programming—algorithms and tools).

Putting your data structure on a diet

Hervé Brönnimann[1] Jyrki Katajainen[2,*] Pat Morin[3]

[1] *Department of Computer and Information Science, Polytechnic University*
Six Metrotech, Brooklyn NY 11201, USA
hbr@poly.edu
[2] *Department of Computing, University of Copenhagen*
Universitetsparken 1, 2100 Copenhagen East, Denmark
jyrki@diku.dk
[3] *School of Computer Science, Carleton University*
1125 Colonel By Drive, Ottawa, Ontario, Canada K1S 5B6
morin@scs.carleton.ca

Presenter. Jyrki Katajainen

Abstract. Consider a data structure $\mathcal{D}$ that stores a dynamic collection of elements. Assume that $\mathcal{D}$ uses a linear number of words in addition to the elements stored. In this paper several data-structural transformations are described that can be used to transform $\mathcal{D}$ into another data structure $\mathcal{D}'$ that supports the same operations as $\mathcal{D}$, has considerably smaller memory overhead than $\mathcal{D}$, and performs the supported operations by a small constant factor or a small additive term slower than $\mathcal{D}$, depending on the data structure and operation in question. The compaction technique has been successfully applied for linked lists, dictionaries, and priority queues.

The full version of this paper will be published in the CPH STL report series. All reports in this series are available online at the CPH STL home page `http://www.cphstl.dk`.

The slides shown at the workshop are also available online at the CPH STL home page `http://www.cphstl.dk`.

[*]Partially supported by the Danish Natural Science Research Council under contract 272-05-0272 (project "Generic programming—algorithms and tools").

The JEWL GUI library, as a powerful teaching tool

Manuel Macías Córdoba

Department of Computing, University of Copenhagen
Universitetsparken 1, DK-2100 Copenhagen East, Denmark
`manuelmaciascordoba@gmail.com.dk`

Abstract. The JEWL (John English's Window Library) is an easy to learn but powerful GUI development tool for the ADA language. This paper covers all aspects regarding the topic proposed on the title, as what is this library, why was developed and by who, what can this Library do regarding GUI development and how; why is this library along with the ADA programming language one of the best choices for academic learning programming. The ADA language and the JEWL library will be put in contrast with the criteria of a foundation language in order to analyze its teaching possibilities; as well as give students feedbacks regarding this subject. Two examples of fully operational GUIs developed by students and implemented with JEWL are given in order to show some work made by students, these examples consist on a basic graphical cinema ticket vendor and an abstract data-structure editor.

Keywords. GUI, library, ADA language, JEWL, learning

1. Introduction

The main goal for this paper is not to look down on the other GUI developer languages and libraries as teaching tools, instead of that, it aims to give reasonable facts that will point the JEWL library as a powerful teaching tool. In fact this library was originally developed aiming at novices, to enable reasonably sophisticated GUI applications to be built with a minimum of effort. In order to achieve this paper goal, a brief introduction to ADA history and features will be done, as well as an extensive explanation of the JEWL library. As this library is part of the ADA language is important to explain why ADA is a good academic learning programming language. The intention of JEWL is to provide a development kit for GUI-based programming in ADA which is sufficiently simple that it can be used from the "Hello world" stage onwards. For this reason the emphasis is on ease of use rather than completeness. Existing GUI packages are often bewildering to novices with the range of facilities they provide, and they achieve flexibility at the price of complexity. As a result, hand-coding a GUI can be extremely difficult. Languages like Visual Basic use a GUI builder to avoid the need for hand-coding, but the code that a GUI builder generates is often difficult to understand (and easy to avoid having to understand). For developing production code this is not an important issue, but in education it can cloud the student understanding of what is really going on. It is also easy for students to get sidetracked into perfecting the appearance of the user interface at the expense of perfecting the desired functionality. JEWL is relatively inflexible by comparison with systems intended for developing production code and only provides access to a limited subset of the underlying facilities, but it is still sufficient for a wide range of novice programs. It is designed so that a program using a graphical interface can be developed by hand-coding, such that the resulting program structure will be similar to an equivalent program with a traditional text-based interface. During the last years the computer science department of the polytechnic faculty of Madrid has used the JEWL library in order to teach their students the basics of GUI development. This paper will include some student feedbacks explaining their experience working with it as well as two completely functional programs developed with the use of this library by them.

2. ADA language: history and features

2.1 Brief ADA history

In the 1970s, the US Department of Defense (DoD) was concerned by the number of different programming languages being used for its embedded computer system projects, many of which were obsolete or hardware-dependent, and none of which supported safe modular programming. In 1975 the Higher Order Language Working Group was formed with the intent of reducing this number by finding or creating a programming language generally suitable for the department's requirements; the result was ADA. ADA was originally targeted at embedded and real-time systems. The ADA 95 revision, designed by S. Tucker Taft of Intermetrics between 1992 and 1995, improved support for systems, numerical, and financial programming.

2.2 ADA features

Notable features of ADA include strong typing, modularity mechanisms (packages), run-time checking, parallel processing (tasks), exception handling, and generics. ADA 95 added support for object-oriented programming, including dynamic dispatch. ADA supports run-time checks in order to protect against access to unallocated memory, buffer overflow errors, off by one errors, array access errors, and other avoidable bugs. These checks can be disabled in the interest of efficiency, but can often be compiled efficiently. It also includes facilities to help program verification. For these reasons, it is very widely used in critical systems like avionics, weapons and spacecraft. It also supports a large number of compile-time checks to help avoid bugs that would not be detectable until run-time in some other languages or would require explicit checks to be added to the source code. ADA's dynamic memory management is safe and high-level, like Java and unlike C. The specification does not require any particular implementation. Though the semantics of the language allow automatic garbage collection of inaccessible objects, most implementations do not support it. ADA does support a limited form of region-based storage management. Invalid accesses can always be detected at run time (unless of course the check is turned off) and sometimes at compile time. Unlike most ISO standards, the ADA language definition (known as the ADA Reference Manual or ARM) is free content. Thus, it is a common reference for ADA programmers, not just programmers implementing ADA compilers.

3. ADA: Teaching language

To be able to decide if a programming language is suitable for academic learning we have to analyze it, taking the foundations of a teaching language as a guide. These foundations are classified into intrinsic elements related to specific language features, and extrinsic factors such as tool support. The analysis was done by Benjamin M. Brosgol. In the following paragraphs I will list the most relevant points of his analysis.

3.1 Intrinsic criteria

3.1.1 Learnability

A foundation language needs to strike a balance between being sufficiently high level to provide a natural notation for modeling a range of software problems, and having an understandable mapping to an underlying target machine and operating system environment. Analyzing the ADA learnability we get to this conclusions: ADA is a large language, and although size by itself does not imply that a language is difficult to learn, in fact the interrelationship of ADA's facilities does present a challenge to a teacher, ADA separates unit specifications from unit bodies, a design decision that supports encapsulation and reduces compilation costs; types in general are a difficult notion for many students to master, and conceptual types introduce an additional level of abstraction; the attribute notation in ADA is fairly complex, and it generally takes students some time before they have an intuitive grasp of what is used where. In conclusion we can say that ADA strikes a reasonable balance between its high-level features and its goal for an efficient stack-based run-time model.

3.1.2 Expressiveness / generality

The language should be powerful enough to express the variety of data structures and algorithmic elements covered in a foundation course. The language should support a variety of development methods, since no one approach is appropriate for all applications. ADA clearly provides the comprehensive features, especially in the realm of data structuring that is so critical in a foundation course.

3.1.3 Encouragement of sound software engineering, including object-oriented programming. Strong typing, separate compilation, and encapsulation should be provided

The language should also have a robust approach to reusability; a good test case is how one can define a reusable module for container data structures such as stacks and queues. With all the attention that object orientation is receiving in the professional software community, universities are increasingly motivated to introduce OOP early in their computer-science curricula. Thus it is useful if a foundation language is object oriented, with support for classes, inheritance, polymorphism, dynamic binding, and related topics. ADA was designed with software engineering as a primary goal. It offers a good support in areas that are more germane to a foundation course, for example providing a standard generic mechanism, stronger type checking, and an efficient stack-based run-time model.

3.1.4 Support for concurrent programming

ADA provides a general, high-level model based on explicit communication (the rendezvous), and a structured approach to mutual exclusion (protected objects), while also supplying lower-level mechanisms and specific scheduling semantics and control that may be needed for real-time and other applications.

3.1.5 API functionality

The API should be structured in a consistent way and make effective use of the language definitional facilities. ADA does offer some standard packages although people familiar with the java API will find it simpler to access from Java than from ADA.

3.2 Extrinsic criteria

3.2.1 Availability of tools

A free ADA compiler based on the GNU gcc technology ("GNAT") was developed at NYU and released at the same time as the language standard. ADA Core Technologies, Inc., continues to upgrade this compiler and makes a public version openly available on the Web. Other vendors are also distributing free or low-cost versions of their ADA products, to encourage their adoption at universities.

3.2.2 Availability of good textbooks

Texts need to be geared to students versus professional programmers, with detailed coverage of basic data structures and algorithms.Feldman cites 17 published books since 1995 with ADA 95 as their focus. Of these, five textbooks are oriented towards first-year students.

3.2.3 Portability/stability/standardization

Although ADA permits implementation-dependent decisions in a number of areas (for example the choice of "by copy" or "by reference" for a formal parameter of an aggregate type), experience with portability of ADA source code has been extremely favorable. Vendors did not rush out with premature implementations of the language, and at least with ADA 95, good quality compilers were available not long after the publication of the standard.

3.2.4 Status as "hot" technology (i.e., where the jobs are)

Given the competition for students and the pressure to teach marketable skills, there are some benefits to teaching a language that is popular in the marketplace. This tends to conflict with the previous goal, however, since the "hottest" technology tends to be the least stable.

4. JEWL: analysis

The previous section has explained why ADA is a good choice for academic learning based on the foundations of a teaching language. JEWL is an ADA library for developing GUI applications, not a language by itself, so most of the criteria stated in the previous section can not be applied to the JEWL library. We will analyce the JEWL library in the criteria established in the previous section, excluding the points were our analysis is not applicable.

4.1 Intrinsic criteria

4.1.1 Learnability

Analyzing the JEWL learnability we get to these conclusions: The JEWL library offers a complete and easy to understand specification, which will allow the students to understand what they are doing from the beginning. This specification is completely documented and its functionality gathers all the possible needs for novice programmers.

4.1.2 Expressiveness / generality

The library allows the programmer to code all kind of user-machine communication such as: point-and-click, graphical, data-input, and data-output communications.

4.1.3 API functionality

JEWL offers a new package for the ADA language, increasing the ADA API functionality.

4.2 Extrinsic criteria

4.2.1 Availability of good textbooks

JEWL documentation is online so it can be accessed by anyone.

4.2.2 Status as "hot" technology

The JEWL library was developed with the novice programmer in mind, and is focused in giving the programmer the possibility to learn how the GUI applications work. These facts establish JEWL applications as not "hot" technology, but as it is explained before, the use of "hot" technology must not blind us when selecting an academic teaching language/library.

5. JEWL library specification

In order to understand the capacity of the JEWL library as a teaching tool is important to know the library specification. In this specification its main features and possibilities are shown.

5.1 Window Type

The window type is an abstract class that provides basic behavior which all windows share. These are some examples of the multiple operations (common to all windows): Show (Window,Visible) Hide (Window)

5.2 Window Subclasses

The primary window subclasses are containers and controls. They share the behavior common to all windows (above) and provide extra behavior as well.

5.3 Containers

Containers are windows which can contain other windows. All windows except frames and dialogs (see below) must be contained within some other container window. There are some restrictions on the types of container that a window can be attached to (for example, a menu item must be attached to a menu). Most windows specify an origin, a width and a height whose coordinates are taken relative to the enclosing container. Positive widths and heights are absolute values, but zero and negative widths and heights are interpreted as being relative to the width and height of the enclosing container. The types of container windows available are as follows: - Frame

Type: A frame is a top level window with a title bar, system menu, minimize and maximize buttons, and a close button. Closing a frame generates a command. Frames are normally visible, but can be hidden if required. A frame should be used as the main window for an application. - Dialog Type: A dialog is a top level window like a frame, but it only has a close button on its title bar. Dialogs are intended for user interaction. When a dialog is executed, it becomes visible and all other windows are disabled. Execution of a dialog continues until a command is generated by closing the dialog window or by clicking on a button attached to the dialog. Dialog windows do not have a menu bar. - Panel Type: A panel is a plain window which is used as a container for other sub windows. - Menu Type: A menu is a pull-down list of items attached to a frame menu bar. The items on a menu are either menu items (which generate a command when they are selected), submenus (which display another menu when selected), or separators (horizontal bars used to separate a menu into subsections).

5.4 Controls

Controls are windows for user interaction. They hold values (e.g. a text string) which can normally be set by the user, as well as being accessed and altered by the program itself. Some controls (e.g. menu items) generate commands which can be used to trigger actions in the program. - Buttons. Buttons are rectangular labeled controls which generate a command code when pressed. "Default" buttons are displayed with a heavier border and respond when the enter key is pressed. - Editbox. An editbox is a text control containing a single line of text whose value can be edited by the user. - Boolean controls. Boolean controls are text controls which can be toggled between two states (checked or unchecked).

5.5 Menu items

Menu items can only be attached to menus. When a menu is clicked, a pull-down menu appears which can consist of menu items (each of which executes a command when clicked) or further menus.

5.6 Checkboxes

A checkbox is a labeled control with a left-aligned box that can be checked or unchecked. Clicking on a checkbox (or pressing spacebar when it is selected) toggles the checkbox between the checked and unchecked states.

5.7 Radio buttons

A radio button is a Boolean control with a left-aligned box that can be checked or unchecked. Radio buttons attached to the same container form a group. Clicking on an unchecked radio button will set the radio button to the checked state and will also uncheck the other radio buttons which belong to the same group (i.e. buttons that are attached to the same container). Unlike a checkbox, a radio button cannot be turned off by clicking on it; we can only uncheck a radio button by checking another button in the same group.

5.8 Multiple controls

Multiple controls contain multiple lines of text numbered from 1 upwards. Individual lines can be accessed by specifying a line number. The user can select a particular line by clicking on it with the mouse or using the keyboard arrow keys when the control is selected. Specifying the line number to access as 0 will access the currently selected line. If no line is selected, the current line number will be reported as 0, but its contents can still be accessed. A Constraint Error will be raised if an attempt is made to access a line beyond the last one.

5.9 Listboxes

A listbox is a list of lines of text (initially empty). The lines are sorted into ascending order by default, but can be left unsorted if required. For a sorted list, the position at which a new line is added will be ignored, with the new line being inserted at the appropriate position according to its value. When no line has been selected, the contents of the current line will be reported as an empty string ("").

5.10 Comboboxes

A combobox consists of an edit control together with a drop-down list (effectively a combination of an editbox and a listbox). The currently selected line is displayed in the edit control, and you can specify whether the user is able to manually edit this value. If not, only the values in the drop-down list can be selected. If the contents of the edit control match one of the values in the list, the position of the corresponding line in the list is reported as the number of the currently selected line. Otherwise, the number of the current line is reported as 0. Accessing the value of the current line (line 0) will report the current value of the edit control.

5.11 Memos

A memo is a simple multi-line text editor similar to Windows Notepad. There are several memo-specific operations in addition to the standard operations on multi-line controls. The user can select a block of text spanning multiple lines using the mouse (or by moving the cursor with the shift key held down) and there are operations to fetch, replace, and delete the selected text, find the line and column position of the start and end of the selected text, and get the text total length (which will include one or more additional end-of-line characters if the text spans more than one line).

5.12 Canvases

A canvas is a blank rectangle for drawing on which can optionally be set to generate a command code when the mouse is clicked on it. A canvas has an associated font (the same as the parent window by default), a background colour (initially white), a pen for drawing lines (initially black, one pixel wide) and a fill colour used to colour closed shapes (initially white). The freedom of expression available with canvases makes these the most complex component of all, with over 20 available operations. There are operations to draw lines, rectangles (with or without rounded corners), ellipses, circles, line sequences, polygons and text. The font, pen size and colour, and fill colour can be changed at any time and will affect all subsequent drawing operations (but everything drawn previously will be unaffected). Rectangles can be given rounded corners by specifying a rounding factor, which gives the X and Y offsets from the corners to the points where the rounded corner begins. Anything drawn on a canvas will normally stay there, but the canvas can be erased or the current state of a drawing can be saved and then restored later (which provides a basic "undo" facility). For example, a clock can be drawn by drawing a circle, saving the drawing, and then drawing the hands. To change the position of the hands, restore the saved drawing (thus erasing the hands) and then redraw the hands in the new position. You can only save one copy of the drawing, so if you save it a second time you will lose the previous saved copy. A canvas can be set up to generate a command when the mouse button is pressed inside its borders. There are operations to get the position at which the button was pressed, to get the current mouse position, and to test if the mouse button is still down, and whether the mouse has been moved. As long as the button is down, the mouse position will be tracked even if the mouse is moved outside the canvas. You can track the mouse visually by saving the drawing when the mouse is first pressed, then restoring the drawing, and drawing a new line from the initial mouse position to the current one.

5.13 Dialogs

- File dialog. File dialogs allows users to select or enter a file name. - Open dialog. Open dialogs allows the users to select or enter a file name for use as an input file. The file name selected must be the name of an existing file. - Save dialog. Save dialogs allows the users to select or enter a file name for use as an output file. If the create parameter to the constructor function below is true (as it is by default) and an existing file is selected, the users will be asked if the file should be overwritten. If it is false and the file does not exist, the users will be asked if it should be created.

6. Student feedbacks

6.1 Alberto García García

JEWL is a good graphical library to start working with because of its API, which is very simple. It has the sufficient methods to give the developed applications the strength and versatility needed. Another point of interest is the scalability of the interface, allowing new functions to be implemented and added in different levels. This scalability begins with the kernel of the interface and upwards, or using the methods the kernel provides. This feature allows the JEWL library to be used by people with few experience in the GUI development area and also by experience programmers, because by introducing new functionalities we can achieve a more powerful interface which will make applications more appealing.

6.2 Jaime Sanchez-Laulhe García-Mercadal

Having working with different GUI libraries, I can say that using JEWL for my first approach was a good choice. The JWEL library allows you to understand all the process of the development of GUI applications in the depth you need, as it has different level possibilities. If you lack the experience and you want to make a GUI software tool you can just work with the instanced methods, the Jewl.Simple windows, which allows you to get right to work on your application. But if you have more experience and want to develop a more sophisticated application you can use the Jewl.windows library.

7. GUI examples

The following applications were implemented by students using the JEWL library, the programs aim was basically to allow the students to learn the GUI development; using the JEWL library the student has the opportunity to make a first approach to GUI development and learn its possibilities in an easy and complete way.

7.1 Abstract data- structure editor

The purpose of this program is to allow a graphical navigation thought abstract data structures, in concrete lists, records, trees, and any combination of those, such as, trees of lists of records, lists of trees, etc. The program is completely graphical. It runs trough windows (error messages, save, delete, open, close, etc) and it opens a main screen where your abstract data structure is drawn, it also allows an in-depth navigation: go forward, backwards and move into and outside levels, and of course it allows modifications to the data structure. Once saved a data structure is stored as a Haskell functional data type, allowing the use of these files in a Haskell program well as being able to load these files so you can modify the abstract data structure graphically.

7.2 Cinema ticket vendor

This program emulates a graphical cinema ticket vendor. It allows you to select the cinema tickets you want to buy by a simple point-and-click interface. The program focuses on the use of point and click communication, as well as graphical navigation trough interfaces; thanks to the JEWL API, designing and implementing this application was done in a reasonable time, allowing students to understand and develop a GUI based on point and-click-communication.

8. Conclusion

This paper has touched all the possible topics regarding our main goal: to explain why is the JEWL library good for academic learning. We have spoken about the ADA language, we have seen its features, history, and we have analyzed the language academic learning potential. Two feedbacks of students from the "Facultad de informatica" have been given along with the explanation of two applications developed with the JEWL library, to give the point of view of computer science students. With all the stated before and with the JEWL specification and an analysis regarding the foundations of a teaching language we can ensure that our goal has been achieved. The JWEL library will help everyone that wants to learn more about GUI development, and will allow them to hand code their applications, allowing the programmer to understand the hole process.

References

[1] Benjamin M. Brosgol, "A Comparison of Ada and Java as a Fundation Teaching Language.," `http://www.gnat.com`.
[2] Wikipedia, `http://www.wikipedia.org`.
[3] John English webpage, `http://www.it.bton.ac.uk/staff/je/`.

Java ME: Huge tool set for the small world

Miguel Fiandor Gutiérrez

Department of Computing, Universidad Politécnica de Madrid
Montegancedo sn, 28660 Boadilla del Monte, Spain
`mfgutierrez@gmail.com`

Abstract. Java Micro Edition Platform (Java ME) is one of the most popular technologies developed for small devices like, pagers, mobile phones, and PDAs. Enterprises like Nokia, Sony-Ericsson, HP and many more, include this software in their devices. This paper presents a short introduction to the different components of this platform, Java programming language, virtual machine, and APIs, and describes some programming developer tools. The main purpose is to explain the reader the internal characteristics of the applications developed using this platform, how to create them, and how to give them communication functionality. Finally some general advantages and facilities of Java ME are given to emphasize reasons of why it has high and extended market influence.

Keywords. Java ME, MIDlet.

1. Introduction

Java 2 Micro Edition (J2ME) is one of the biggest platform freely available in the Internet for developing applications will run on a group of devices with common characteristics, as small memory, limited power, constrained user interfaces, etc. Thanks to the high amount of free information, manuals, documents, forums, are easy available in the Internet, it can be used by final customer users as well as international enterprise developers. A short description on the main components of this platform is given below in order to present afterwards the internals of this tool set, and how many different applications can be easy developed. The main components of the platform are the following:

- Java programming language is one of the most popular object-oriented programming languages. Its main characteristics are, its similar syntax to C and C++, its platform independence (the same program can be run in different platform without any changes), and automatic garbage collector (memory management has not to be done manually, Java runtime takes care about memory leak problems).
- Virtual machine (VM) works between the Java byte code generated by a compiler and specific hardware of any machine. That is why some versions of virtual machines exist, they are related with the hard-

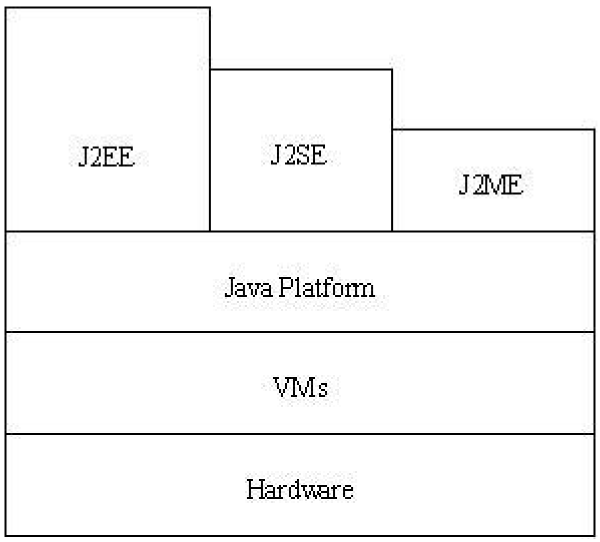

Figure 1. Example of different layers of Java Platform and how integrates into a system.

ware platform you will run your application. Platform independence is thanks to this component.

– A full set of standard application programming interfaces (APIs[1]). J2ME, as its brother versions, J2SE and J2EE, comes with a high amount documentation API being in this case the shortest one. It is actually a subset of J2SE API because in small devices with less capabilities is not needed all the functionality that incomes in others Java platforms.

To understand how J2ME works we have to know well what do Configurations and Profiles mean. Configurations specify the Java libraries and VM capabilities that are present in a device that implements such configuration, in other words, the minimum J2ME platform for a family of devices. At the present time, CLDC[2] (Connected Limited Device Configuration) is an existing configuration that includes devices in a group with the same or similar characteristics. An example of characteristics specified by CLDC is the following:

– At least 192 KB of total memory budget available for the Java platform
– A 16-bit or 32-bit processor,
– Low power consumption, often operating with battery power,
– Connectivity to some kind of network, often with a wireless, intermittent connection and with limited bandwidth.

On the other hand, a Profile[3] is a set of APIs that bound to a Configuration; they offer together functionality to a specific device. The Java Community Process[4] (JCP) defines these Profiles, and tested to insure the platform independence of any application. JCP is the Community Development of Java Technology Specifications; firstly they receive a Java Specification Request (JSR) from experts of a group of companies interested in building that specification. Then the process continues through different states until JCP creates it, so every specification is linked to a JSR.

2. Software Tools

The development of many applications for the small devices I am focusing in this paper is due to free software tools[5] are available in the main website of J2ME. The most important ones are explained below:

- Sun Java Wireless Toolkit 2.5 is a set of tools for developing, emulating, testing, and debugging Java applications that will be run on devices with the Java Technology Wireless specification. It has a simple interface for creating new projects and also brings many examples, both can be run over three different devices, two mobile phones and a qwerty device. Characteristics like network configuration, memory storage, security, Bluetooth properties can be selected as developer likes for a better testing phase. It is based on Configuration CLDC 1.0 and Profile MIDP 1.0.
- Netbeans Mobility Pack 5.0 is a tool for developing applications that run on mobile phones, similarly to Sun Java Wireless Toolkit, this one integrates MIDP 2.0 and CLDC into the Netbeans IDE 5.0, which a powerful tool for developing any kind of Java applications. The difference in this case to develop a mobile application instead of any other kind with the Neatbeans IDE 5.0 is to choose a mobile type project at the time the application is created. It has a sophisticated interface that allows you to write, test and debug applications in a really easy way.
- Java Device Test Suite provides a test manager and several tests to reduce time wasted in testing and debugging applications for a device classified in Connected Device Configuration (CLDC) and the Mobile Information Device Profile (MIDP) types. The Java Device Test Suite helps building applications which can ensure their robustness succeeding these tests, and also many others that can be easily developed and standardized for testing future applications.
- And big enterprises like Nokia, Motorola, Siemens, SonyEricsson and others, have their own supplement toolkits for developing applications targeted to their own devices.

3. MIDlets

3.1 What is a MIDlet?

MIDlets are applications designed using the specification MIDP, in other words, applications designed to run on wireless devices with Java Technology, low memory storage and graphics capabilities and processor, such as cell phones or PDAs. In these devices are not present any command shell for executing applications, as substitute exists software in charge of manage every action related with MIDlets, it is called Application Management System (AMS). This software is saved on the device and allows us to execute, pause or stop any application J2ME, in this case MIDlets. AMS also has to manage the life of any MIDlet in the system, and to control the different states could go through a MIDlet while it is being executed.

A MIDlet goes through five different phases: localization, installation, execution, update, and deleting. Localization is when we search for an application using an Internet browser or the AMS in the cell phone. Booth ways have to be able to download or save the application in our device. During the installation the AMS has to inform the user about the installation process state, and catch any exception due to any problem, for example: it is trying to install an older version of an existing MIDlet, and report it to the user too. When the user executes the MIDlet, AMS has to control the different running program states. The update phase consists on once a MIDlet has been downloaded we have to be reported if such version is an update of an existing one, or a new and completely different MIDlet. And if a MIDlet will be deleted, it will be done by the AMS, and AMS will notify about any uncommon result.

In order to give more detailed information about a MIDlet and understand how it works, it is necessary to explain the package javax.microedition.midlet. This package defines MIDP applications and the behaviour of the system. There are two classes inside this package, MIDlet class (MIDP applications), and MIDletStateChangeException class (reports of an error in a status change).

3.2 MIDlet Package

This package contains two classes, MIDlet class and MIDletStateChangeException class. The MIDlet class offers different methods to establish communication between a MIDlet and the system is running over or vice versa. All methods can be divided into three groups:

- The constructor call, which creates a new MIDlet.
  ```
  protected MIDlet ();
  ```
- Methods used by the AMS running in our system to control such MIDlet.
  ```
  protected abstract void startApp () throws MIDletstateChangeException;
  protected abstract void pauseAPP ();
  protected abtract void destroyAPP (boolean incon) throws
    MIDletstateChangeException;
  ```
- Methods used by the MIDlet to notify an event to the AMS running in our system.
  ```
  public final void notifyPaused ();
  public final void notifyDestroyed ();
  ```

MIDletChangeStateException class has only one method that launches an exception when an error in a MIDlet status change occurs.

```
public class MIDletstateChangeException extends Exception;
```

3.3 MIDlets Structure

After learning the different states of a MIDlet and its runtime period is really important to learn its basic structure. This will help us to understand how a MIDlet works, and write our first one will be much easier. MIDlets have the following structure:

```
    import javax.microedition.midlet.*
    public class MIMidlet extends MIDlet
        //Constructor, who has to initialize the variables
        public MiMidlet () {
        }
        //Code to be executed when the MIDlet will be activated
        public startApp () {
        }
        //Code to be executed when the MIDlet will be paused
        public pauseApp () {
        }
        //Code to be executed when the MIDlet will be destroy
        public destroyApp () {
        }
    }
```

3.4 MIDlets Communication

The possibility of being always connected using a small device includes many others applications we could not develop in any other conditions. That is the reason because it is extremely important the use of communication API in J2ME. MIDlets use the packages javax.microedition.io and java.io included in the API of J2ME. First one contains several classes that allow building and management of different network connections: HTTP, datagrams, sockets,... On the other hand, java.io package offers service to read from and write into these connections. The Network connection classes are called Generic Connection Framework, and before describing the principal characteristics of most important classes allocated in javax.microeditino.io, we mention that offers only a public class Connector, main one, which hides every implementation detail. Not all classes and methods are described because that would not follow the intention of this paper. The following graph helps to understand the hierarchy and how a user access to the communication package functionality.

– Interface Connection is in the top of the interfaces hierarchy of Generic Connection Framework. This is the most basic type of generic connection, and every interface inherits from here. Every call creates and opens a connection, each one with different parameters or types, some invocations can be the following:

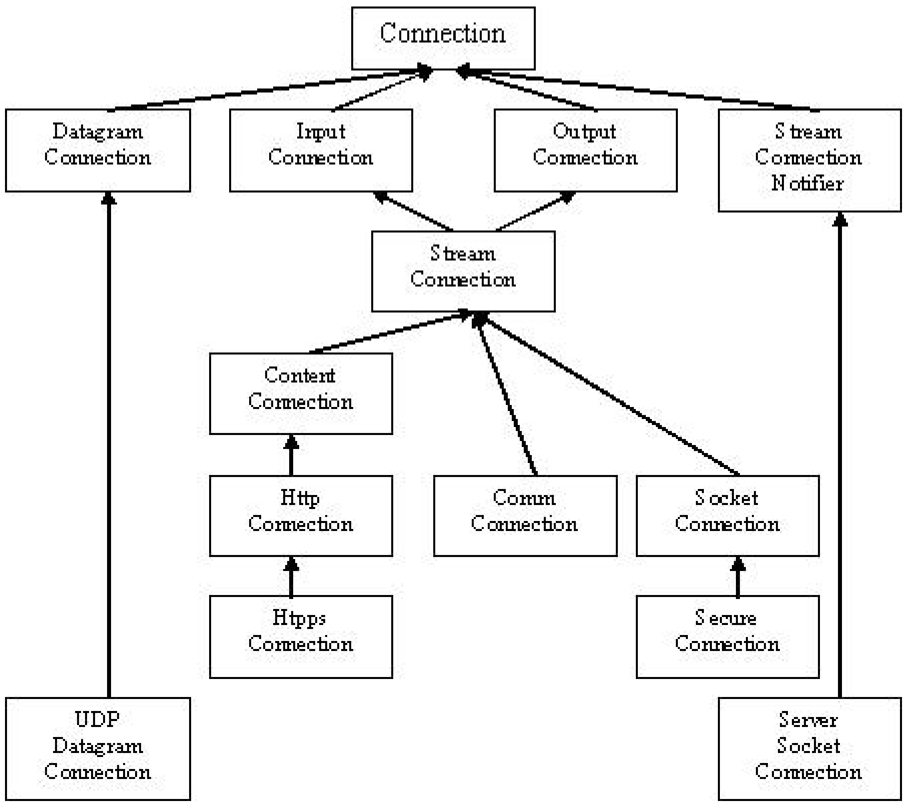

Figure 2. Hierarchy of java.microedition.io package.

```
Connector.open ("http://localhost/index.html");
Connector.open ("socket://anywhere:0000");
```

– Interface StreamConnection represents connections based on incoming and outcoming streams. Where an application will call read and write methods to send or receive data. Also there exist two versions of classes that implement this functionality separately, InputConnection and OutputConnection.
– Interface HttpConnection, necessary methods and constants for an HTTP connection are in this class. The methods presented are only a subset of a long list that allows any kind of operation in this protocol based on request-response, HTTP protocol.
– Interface HttpsConnection includes methods that allow to establish a secure network connection. As the first line indicates this class inherits every method and constant from the interface HttpConnection and others.
– Interface SocketConnection defines a connection between sockets that use streams in their communication. A generic connection with the socket includes a host (String) and a port (int).

- Interface SecureConnection defines a secure connection between sockets. This connection is build invoking the next method:

  ```
  Connector.open ("ssl://host.port");
  ```

  If a secure connection cannot be built due to certificate errors a CertificateException is thrown.
- Interface ServerSocketConnection includes methods for creating a connection with a socket server omitting the host in a generic open call, for example:

  ```
  Connector.open ("socket://:1234");
  ```

3.5 Different applications of MIDlets

MIDlets has been design for many different purposes. Many of them belong to game section, but we can found examples for graphic animations, Internet communication and other utilities. Games are obviously developed just for the user entertainment finding all types of games like arcade, fighting, platform, sports, multiplayer (this type can be considered a communication MIDlet as well). Graphic animations are though to prove their graphic capabilities in a system with poor resources, testing mathematician algorithms, and the colour screen possibilities that offer such device. We can found utilities like: alarms, calendars, world time, currency converters, microreaders (text editors), periodic tables, dictionaries, calculators, and a long list continues really far.

Where this paper is more focused on, the Internet and communication MIDlets, we have browsers, email clients, fax clients, instant messenger chats, database managers, telnet clients, and some more. They use different technologies to connect to servers or other mobiles phones. For example a Bluetooth browser that explores the surrounding Bluetooth devices, chat messaging by IrDA, and in PDAs are fast growing the number of applications that use wifi technology to dialog with any device or server. A common purpose is run a MIDlet that allows to communicate your mobile phone with any computer at home, usually by Bluetooth, and then synchronize them. Also interact from the mobile phone to a computer. With the introduction of wifi technology into mobile phones and PDAs and these extended networks, we can easy interact from really far to any computer, and start thinking in huge amount of possibilities.

It would be nice to include and explain the code of a MIDlet which communicates with other devices by Bluetooth, but it has a huge amount of lines that would make this paper out of range. Instead of including the code a good solution is to check any example that comes with the Sun Java Wireless Toolkit. Also these examples can help us when we are writing our own MIDlet.

4. Advantages of using J2ME

Develop wireless applications for mobile phones or PDAs, or maybe any technology device, but specially this two, demands to be up to date in what is happening, new offers from each company, new hardware, new services and society requests... And this can only be afford by a complete community working in those purposes.

It is difficult to compare J2ME with others existing platforms, there are mainly two reasons. First one, other platforms are not composed of the same elements as J2ME (programming language, Virtual Machine, APIs), and second one, they have different purposes.

In the first group we find Windows Mobile Platform[6] and Symbian OS[7], basically they are operating systems for mobile phones, Pocket PCs, or Smartphones. Applications or programs can be developed for these operating systems in different programming languages, so they also have APIs, software tools, and documentation. But the main reason is those applications will have quite less portability in compare with the applications developed with Java ME. In fact, Windows Mobile Platform and Symbian OS can bring a Virtual Machine installed, what means applications created with Java ME also will run over these operating systems.

In the second group, I have found OpenTrek Platform[8]. It is a platform just for developing interactive network games on mobile devices, at the moment does not work on mobile phones, just on Pocket PCs. The programming language used is C++, also very popular. This platform has the similarity that brings API to develop the games, but the differences with Java ME are clear, and the advantages of using Java ME are listed below.

- Brings the biggest help in this area to application developers, not just refer to high amount of free documentation, manuals, tips, forums, mailing lists, etc, also to all the powerful software tools are freely available. Now anyone a bit interested in this small world can easily become a hobby programmer without any previous knowledge, and it has to be consider that nowadays almost everyone has own mobile phone, and number of PDAs in the market are increasing quickly.
- All applications are run over a safe platform, J2ME. Every Java code is executed within a Java Virtual Machine, which provides a safe environment. The worst case could bring down the VM, not the personal device itself. This condition fixes the problem that could arise from running applications made by non-professional programmers, we are safe about our operating system will be saved, and therefore our device too.
- One of the best advantages is that every single application can be run on multiple devices, all those devices with common Configuration and Profile, it is the awesome portability of Java Platform. Another benefit of portability is the easy sharing of applications via OTA (over-the-air), OTA is the service that allows to download applications by an Internet connection which increases the list of facilities for any kind of user.

- Java Community Process is an existing community that constantly takes care about improvements and updates to the Java Platform. Therefore holds this company at first line of permanently changes that suffer the technology area. This success is due to worry about the technology companies whose requests make their products up to what society demands too.
- Java ME provides a useful and extended API, Java APIs can be the most well organized APIs of every programming language. Every class comes with a short description, a field summary, constructor summary, method summary, and a field detail. This really helps any kind of programmer.
- The mobile phones and technology enterprises have the possibility of take advantage of the portability that Java Platform offers, at the same time they build special support for developing applications focused on their own property devices. What gives them chance to be more independent in this development, and be safe of, for example software copies, or could make their own quality tests.

5. Conclusions

The Java Platform Micro Edition, J2ME, is a perfect example of how a technology development community should be, and it is pronounced in its world business success. Many international companies trust in this technology. To obtain this confidence is not an easy job, J2ME has to show itself as a powerful tool in each different branch at the same time it seems a solid technology group, really well work by JCP, Java Community Process. This is converted into a great market influence what also means popularity and society succeed.

References

[1] API J2ME, `http://java.sun.com/javame/reference/apis/jsr139/`
[2] CLDC, Documentation about Connected Limited Device Configuration, `http://java.sun.com/products/cldc`
[3] MIDP, Documentation about Mobile Information Device Profile, `http://java.sun.com/products/midp`
[4] JCP, The Java Community Process home page, `http://jcp.org/en/home/index`
[5] Java ME downloads, website with links to the most important software tools and documentation about J2ME, `http://java.sun.com/javame/downloads/index.jsp`
[6] Windows Mobile Platform, `http://www.microsoft.com/windowsmobile/developers/default.mspx`
[7] Symbian OS, `http://www.symbian.com/`
[8] OpenTrek-Platform, Pdf document describing the Cross-Platform, `http://www.viktoria.se/fal/publications/2003/mobilehci2003-opentrek.pdf`

Iterators for the trie data structure

Kasper Egdø

Datalogisk Institut, Københavns Universitet
Universitetsparken 1, DK-2100 København Ø, Danmark
`egdoe@diku.dk`

Abstract. A trie is a tree structure used for storing strings over an alphabet and optionally per entry satellite data, and performing fast lookups, insertions and deletions. In this paper some strategies are considered which have different space and time requirements and which extend the data structure with constant size STL-style iterators and provide lookup of the strings from the iterators. Additionally, a more compact DAG-based represention which still allows satellite data, but does not allow updates, is described along with a method to generate the representation.
Keywords. trie, iterators, STL

1. Introduction

The trie data structure is quite old (see, for example, [4]), but the basic operations on the trie do not satisfy all of the constraints that users of the C++ standard library [5] have come accustomed to, in particular the use of iterators to both enumerate entries in the trie, and to have constant size "pointers" to them. Several possible representations for introducing iterator support to the trie are presented here, with the aim of not bloating the nodes of the trie more than necessary, even at the cost of some runtime performance.

A trie uses quite large amounts of memory since every letter of the keys are stored in individual nodes; if one is willing to forego the insert/delete operations on the trie, it can instead be represented by an directed acyclic graph (DAG), which for some applications (e.g. storing dictionaries) can substantially reduce the number of nodes required. Normally this also means the loss of the ability to store satellite data for every entry; here an approach is given that reenables this for the DAG representation. Finally, a practical test is performed with a $268,984$ word dictionary showing that converting it to the DAG representation reduces the number of nodes by a factor of approximately 5, and that this representation may actually be smaller than storing the words of the dictionary explicitly.

The trie data structure has applications in among others routing [3], dictionary storage and spell checking [2].

2. The trie data structure

The *trie* (from re*trie*val) is a tree-based data structure used for storing strings over an alphabet A together with optional per string satellite data. It is thus an associative array with the strings as keys. All strings sharing a common prefix are represented by nodes descending from a common root (see Figure 1). Each element (referred to as a *letter* l) of a string is given on an edge in the graph. Each `(key, value)` pair (or in the case of no satellite data, just the `(key)`) is referred to as an *entry* in the trie. Note that the *depth* of the tree is equal to the length of the longest string stored. A compact representations called a *compact trie* joins nodes with only one child with their child, i.e. storing substrings instead of single letters on edges of a trie and ensuring that every internal node has at least two children. Though feasible to use the compact trie representation with the methods in this paper, it will not be considered further.

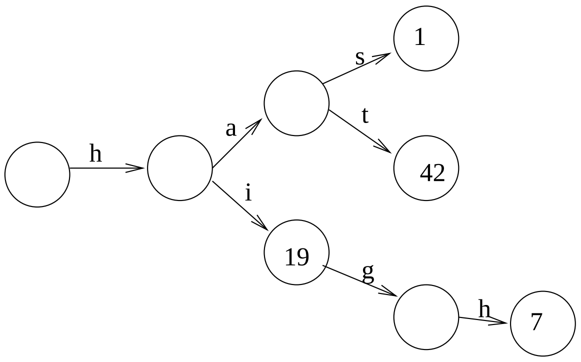

Figure 1. A conceptual drawing of a trie. The entries `(has,1)`, `(hat,42)`, `(hi,19)` and `(high,7)` are stored.

2.1 Operations on tries

The trie allows several operations:

- `insert(entry)`: Insert a new entry into the trie. This is done by recursive descent on the nodes of the trie while consuming a letter from the input string for each matching edge in the trie, effectively matching the prefix of the entry's key. The remainder of the string is then inserted by creating new nodes and linking them into the trie. The runtime is directly proportional to the length n of the entry's key.
- `delete(key)`: Delete an existing entry from the trie based on its key. This works by finding the path through the trie corresponding to the key of the entry to be deleted, and removing the leaf node found. If any nodes on the path from the root end up with no children, they too are removed. The runtime of this operation is also $\Theta(n)$.
- `find(key)`: Look up the entry for a given string. This is done by traversing the tree as in insertion, and if a path corresponding to the search string is found, return a pointer to the value in the trie (or true in the case of no satellite data).
- `prefix-match(key)`: Like `find` but instead of returning the value, return instead a pointer to an internal node in the trie corresponding to the prefix, if one exists.
- `list-children(node)`: Allows iteration of the children or outgoing edges of a node returned from `prefix-match`.

2.2 Relevance to the C++ standard

In terms of the C++ standard library [5], the trie structure is close to matching the capabilities of the `std::set` or `std::map` containers, but with different runtime cost for insertion, deletion and lookup. In addition the trie lacks iterators; the partial aim of this paper is to provide them.

Note that iterating over the children of the individual nodes in the trie is trivial (this is the primary mechanism used by all the trie operations). This is referred to as *node iteration* and the iterators for this are referred to as *node iterators*.

As an aside it is worth noting that while the `std::set/map` uses *equivalence* to determine if two keys match, the trie uses *equality* of the individual letters to determine if two keys match. This and the fact that the keys are not stored explicitly means that implementing `std::map` using a trie is not possible.

Extending the trie to support multiple entries with the same key is easily done by letting the stored value be a linked list of values.

One of the reasons for adding iterators is to allow the string to be retrieved given a node (an operation not supported by the trie). This is practical when building data structures which require reverse lookup (that is from the value to the key) — this way the string can be represented by an iterator and does not need to be stored explicitly. It is not intended (nor optimal) for enumerating all of the strings in the trie — using the node iterators is the correct way to do this.

2.3 Representation

There are several ways to represent a trie. The traditional way is to let each node contain an array of length $|A|$ of pointers to nodes. Nodes corresponding to ends of words have an additional, possible null-valued, pointer to satellite data or, in the case of a trie with no satellite data, a Boolean indicating if this node is the end of a word. This representation has the advantage of giving constant time access to a desired child node, since the position in the array corresponds to the letter of the alphabet. However this can be quite wasteful in the quite common case where most entries are null.

An alternative is to use a resizable array to store the child list. Each entry in this array then stores a letter and a pointer to a child node. This way the space waste is kept to a minimum (however the letters must be explicitly stored), but the constant time access to children is lost. Instead binary search (giving $O(\log_2 |A|)$ access) can be achieved if the array is kept sorted. Unfortunately, insertions and deletions now require reallocation and copying of the array, which makes this an unattractive option.

A (balanced) tree could instead be used for storing the children. This would also give $O(\log_2 |A|)$ access. The disadvantage is that storing such a tree would require storing at least two pointers per child.

Alternatively a representation based on linked lists (according to [4, p.495], first suggested by de la Briandais) can be used. Each node in the trie consists of three elements: `letter`, `pointer-to-next-sibling`, and `pointer-to-first-child`. A leaf in the tree (a node having null values in both `next-sibling` and `first-child`) corresponds to an entry; note that the value of the letter is undefined in this case. Looking up a child now becomes an $O(|A|)$ operation since the linked list must be traversed to find the proper child node. Insertion, deletion and lookup of strings become $O(n|A|)$ operations.

In this paper a variant of the linked list approach is used, the difference being that a special letter θ from the alphabet is used to mark the leaf nodes; for the quite common case of storing null-terminated strings the value θ is the null character, which makes this a reasonable approach. In cases where there is no natural letter to be used for this purpose, the alphabet is extended to A' (in practice by adding a boolean flag). This lets us (ab)use the `first-child` pointer (which should be null for leaf nodes) to point to the satellite data. Additionally, a “sentinel” root node having an undefined letter value is used. This lets us represent the empty trie without special cases. Note that in this representation the letters are not stored on the edges of the tree, but in the nodes (see Figure 2).

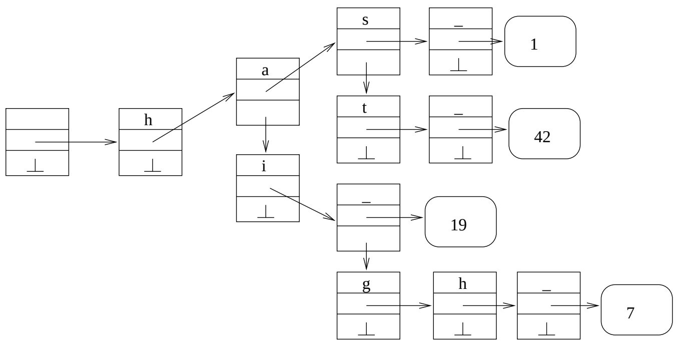

Figure 2. The linked-list representation. The entries `(has,1)`, `(hat,42)`, `(hi,19)` and `(high,7)` are stored.

3. Iterators

In addition to the node iterators described above, we also wish to have entry iterators. These iterators should let us iterate over the entries and retrieve/alter the satellite data, but they should also give us a way to reconstruct the string for the entry they refer to. With normal STL-containers (e.g. `std::map`) the iterator (perhaps indirectly) points to a data structure containing both the key and the satellite data. In the case of the tries it is not possible to point directly to the key since its representation is split over many nodes. Storing the string in its entirety along with the satellite data is of course an option but would be quite wasteful. Since we want the iterators to be constant size we cannot store the strings in these either.

The operations we wish to support for the iterators are the following:

- `increment`: Point to the next entry. This operation should give the no-throw guarantee. In C++ terms, we want both the pre- and postfix operations.
- `decrement`: Point to the previous entry. Also no-throw operation; both pre- and postfix versions.
- `dereference`: Return a reference to the satellite data. No-throw operation.
- `getkey`: Return (a copy of) the string corresponding to the entry. Since this requires construction of a string, this operation can potentially fail.

With the exception of the `getkey` operation, these operations are supported by other STL-style iterators. The dereference operation differs from the normal version by not providing access to a key. In addition we also wish to provide the standard STL iterator operations: default and copy constructors, iterator assignment and (in)equality comparison functions. The `getkey` operation is not supported by the standard trie data structure.

In addition to just providing the operations, it is desirable to do so without increasing the size of the data structure more than necessary and to keep the iterators small. Constant time operations on the iterators are desirable as well, but is not achievable in all cases.

While modelled on the STL-iterators, the concepts and the extensions presented here are equally applicable in other languages/frameworks.

4. Extending the data structure to support iterators

As noted earlier, the main challenge with supporting iterators is to be able to look up the stored string from the iterator refering to a given element. There are two basic ways to do this: the first is moving from the leaf node representing the entry towards the root, and collecting (in reverse order) the letters of the string. The second method requires a way to start from the root and "know" the way to the correct leaf; this method described in this paper for doing this depends on storing for each node the number of entries that it is a prefix for. These methods are described in greater detail in the following.

4.1 The leaf-to-root methods

4.1.1 The brute force method

The leaf-to-root methods require some mechanism for finding parent nodes in the tree, and for finding the next entry when incrementing/decrementing the iterator. The most obvious method is of course to store for each node a pointer to its parent, and for leaf nodes pointers to the previous and next entries. This way an iterator to an entry can exist as a pointer to the leaf node. Retrieving a string of length n is then simply a matter of, in $O(n)$ time, traversing the parent pointers and collecting the letter from each node, and incrementing/decrementing the iterators is simply, in constant time, following the predecessor/succesor pointer. The disadvantage is of course having to store 3 pointers for all leaf nodes (thus more than doubling the size of the nodes), and that the iterators do not have the random-access property.

4.1.2 Eliminating the predecessor/succesor pointers

The predecessor/succesor pointers can be eleminated. Instead decrement can be implemented as follows. The current node x is set to a leaf node. The parent node y is found, and the child list is traversed until the node z before x is found. If z exists (x is not the first child of y), the last leaf node from z is found by recursively following the last child of each node until a leaf node is found. This leaf node becomes the new iterator value. Is z does not exist, the algorithm sets $y \leftarrow x$, and restarts.

The runtime cost of an iterator decrement depends on the number of siblings nodes looked for, and on the number of last child nodes followed. Finding (or failing to find) a node's previous sibling is bounded by the alphabet size. In the worst case, the number of siblings to be looked up is equal to the length n_1 of the current string. This takes time $O(|A|n_1)$. Looking up the last children of the new string of length n_2 takes time $O(|A|n_2)$. Thus, the worst-case cost of any given iterator operation is a function of the length n of the longest string and of the alphabet size: $O(|A|n)$.

Iterator increment is cheaper, since finding the next sibling can be done in constant time (via the `next-sibling` pointer). If no next sibling exists, the parent node is chosen, and its next sibling is attempted used, and so on. When a next sibling is found, the `first-child` pointers are followed recursively until a leaf is reached. The number of pointers followed is at most twice the height n of the tree (in the worst case we traverse a path of length n from the current leaf all the way to the root, and a path of length n to the desired leaf). This is then an $O(n)$ operation.

Only a single extra pointer (the parent pointer) is stored for iteration. The iterators do not have the random access property.

4.1.3 Using the linked list

When the children of a node are stored in a (singly-) linked list, it is even possible to avoid storing a parent pointer in each node. Instead the final node in the list, which would usually have a null value as its next-pointer, is changed to point to the *parent* node instead. This requires a special case for the root node since it has no parent — it can either point to itself, or be null (see Figure 3).

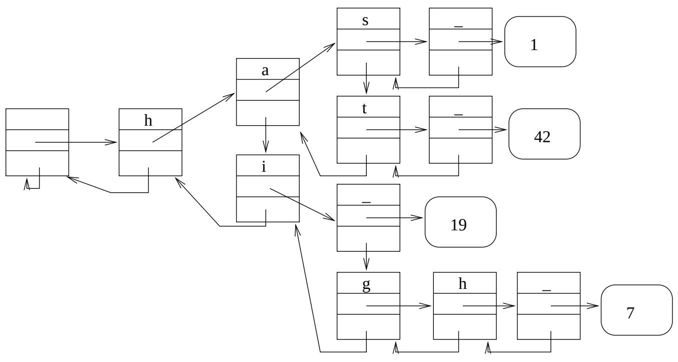

Figure 3. The linked-list representation using the last `next-sibling` pointer to point to the parent. The entries `(has,1)`, `(hat,42)`, `(hi,19)` and `(high,7)` are stored.

The implementation of the node iteration is actually trivial to adjust. Instead of having a `(first-child, null)` pair as begin and end, the pair then becomes `(first-child, this)`. However, finding the parent or next sibling of a node is no longer trivial, since the next pointer may either refer to the next sibling *or* to the the parent.

The next sibling for a node x can be found (or determined not to exist) by looking at the node y given by x_{next}. By iterating over the children of y we can determine that y is x's sibling if x is not in the child list. Conversely we can know if it is x's parent. Determining the next sibling node thus requires $O(|A|)$ time.

Based on this we can find the parent of a node x by finding all of the siblings following it. At this point the last siblings `next-sibling` must point to the parent. This requires $O(|A|^2)$ time. When we determine that a node y is the parent of x, we can remember the previous sibling of x. Thus finding the previous sibling is an $O(|A|^2)$ operation.

Given the cost of finding parents, retrieving a string of length n takes time $O(|A|^2 n)$.

Since the cost of finding previous and next siblings increases by a factor of $|A|$ compared to the version having parent pointers, the costs of increment and decrement increase as well. They become $O(|A|n)$ for increment and $O(|A|^2 n)$ for decrement.

This approach lets us retrieve the strings with *no* additional iterator storage cost. The iterators do not have the random-access property.

4.2 The counting method

The counting method requires that for each node in the tree, we associate (and store along with) a number c which represents the number of leaf nodes that this node is a prefix of. The leaf nodes have a value of 1. The data structure is illustrated in Figure 4. This way we can implicitly number each entry in the tree, beginning with 0. An iterator is then simply an integer corresponding to the number of an entry. Looking up an entry n is done by skipping the preceding n entries. This is a recursive process beginning at the root node. For each child the c value is retrieved, and if it is less than or equal to n, c is subtracted from n, and the next child considered. If the value c is greater than n, the child becomes the current node. When n is 0 and a leaf node is found, this leaf node is then the desired leaf node. During the recursion, the letters from the visited nodes together constitute the string corresponding to the entry. Note that this method requires that all leaf nodes are the first child among siblings.

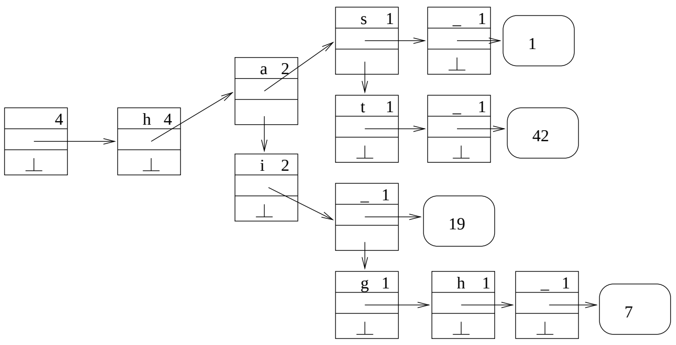

Figure 4. The counting method in which each node is marked with the number of entries that hang of it. The entries `(has,1)`, `(hat,42)`, `(hi,19)` and `(high,7)` are stored.

Looking up an entry using the counting method has the same runtime complexity as a normal lookup, since the only difference between the two is whether the c value or the letter value is examined. The storage used for each node is increased with the space for one integer, independent of how the child lists are represented.

Some maintenance is required in order to preserve the correct values of c during insertion and removal of entries. When inserting a new entry, every existing node in the path has its c value incresed by 1; and likewise when removing a node, they are decreased by 1. Note that when an internal node has its c decreased to 0, it should be removed (since it has no entries hanging off of it). This maintenance does not effect the asymptotic runtime of insertions or deletions.

As noted earlier, when using the counting method, an iterator simply consists of the number n. This enables us to construct random-access iterators. However, in order to be able to dereference an iterator, we also need it to store a pointer to the container (the trie). Additionally we have the somewhat peculiar trait that dereferencing an iterator is not a constant-time operation. Alternatively we can forego the constant-time operations on an iterator and additionally store a pointer to the leaf node of the entry, and update it whenever the iterator changes. Either way, looking up the string of an entry is still a non-constant time operation.

The problem with this method is that when inserting or deleting an entry i which becomes or was the n_ith entry, all iterators with $n \geq n_i$ are invalidated. However, as we shall see in Section 5 this representation is quite useful for immutable tries.

5. DAG representation of a trie

While a trie manages to reuse prefixes of strings, no such reuse of equal postfixes is achieved. If we are willing to forego insert/delete operations on the trie, we can instead represent it as a (minimal) directed acyclic graph (see Figure 5). This normally means that we lose the option of having satellite data — however in Section 5.2 it is shown how this can be achieved via the counting method.

To understand intuitively why using shared postfixes is usable, consider the case of storing a dictionary of Danish words in their various conjugations. Most nouns have conjugations ending with "en" "ens", "er", "erne", "ernes" and the empty suffix. In the trie each word (e.g. "person" and "hest") would have to have an explicit subtree to represent each of these sets. Instead we wish to represent this by a single shared node. Of course, this is not limited to reusing preselected sets of suffixes — we wish to exploit all suffix equality.

We have several different options for representing the DAG. Since insert/delete operations are disallowed, using linked lists is not as necessary as earlier, so we can store in each node a letter, and an array (with a length) of pointers to child nodes.

The runtime costs for lookup of a string is the same as for the linked-list trie, i.e. $O(|A|n)$.

The number of nodes in a DAG is at most the same as in a corresponding trie. The actual number of nodes does not depend directly on the number or length of strings stored; it also depends on the "complexity" of the set of words being stored — sometimes adding a string can *reduce* the number of nodes.

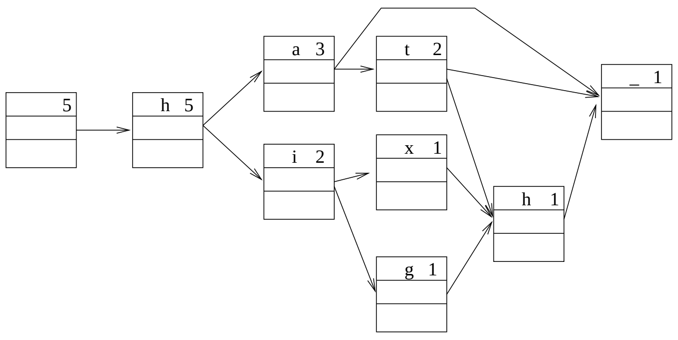

Figure 5. The DAG version (no satellite data). The entries `(ha)`, `(hat)`, `(hath)`, `(hixh)` and `(high)` are stored.

5.1 Generating the graph

In the following a method is described for creating a DAG from an existing trie. First all child lists of all nodes in the trie are sorted according to their letters. The specific order is not important, merely that all child lists are sorted the same way, and that the θ values are stored first. Each node in the DAG consists of a letter and an array of pointers to children. For the creation, a set D (in C++ a `std::set` is a prime candidate) of created DAG nodes is used. Initially a single childless terminating node t with the special θ letter is inserted into the set D. Each node a in the trie is then mapped to a node b in the DAG. The leaf nodes in the trie are trivially mapped to the t node. For all non-leaf nodes a in the trie, their children $a_1 \dots a_n$ are recursively mapped to $b_1 \dots b_n$. After mapping the children of a, a new DAG node b with a's letter and the $b_1 \dots b_n$ child list can be constructed. If there exists a node $c \in D$ equal to b, c is chosen as the mapping of a and the newly constructed b is discarded. Otherwise, b becomes the mapping of a and is also inserted into D. Note that testing DAG nodes for equality requires only comparing the letter and the pointers of the child lists, and is thus an $O(|A|)$ operation. When the root node a of the trie has been mapped to a node b in the DAG, the construction of the DAG is complete, and the set D can be destroyed (though not its contents).

For every node in the trie the total work done consists of a lookup or insertion in the set D. Assuming a red-black tree is used for storing the set D, a lookup in the set takes time $O(|A| \log |D|)$. Since $|D|$ is bounded by the number of nodes N in the trie, creating the trie is an $O(|A| N \log N)$ time and $O(N)$ space operation.

5.2 Adding satellite data

Here only the unique-entries solution is considered, the multiset version is described later. Storing satellite data is achieved by adding an array with all the satellite data values in the same order as when traversing the DAG. Via the counting method described earlier (adding a count of leaf nodes hanging from each node), we can use indexes into the tree to look up both strings and their values, and we can retrieve the index of a given string.

5.3 Iterators

Using the counting method, the iterators are implemented the same way as in the counting method for the normal trie (a number, and a pointer to the container). In C++ terms it is tempting to also add to the DAG trie an `operator[](int)` to allow indexing into the container by index.

5.4 Multisets

Implementing multisets, while still keeping the iterator to a single number, can be achieved by introducing more than one terminator node. When constructing the DAG from a trie, first the maximum number x of entries with the same key is determined. Then, instead of a single terminator node, x terminator nodes $\theta_1 \dots \theta_x$ are constructed, with $c = 1 \dots x$. When constructing the DAG, instead of choosing the single terminator node, the proper θ_x is chosen for each leaf in the input trie.

From an index it is still trivial to retrieve the data (the index is still an offset into the array). From the index the string can be looked up as well, the difference being that a leaf terminator node can be encountered before the index variable has been decreased to 0.

When looking up for a string and one of the leaf nodes θ_i with $c = i$ is encountered, instead of the normal offset o in the data array, a range $[o \dots o + c)$ is returned.

Note that the number of terminator nodes is bounded by the number of identical strings, and thus by the total number of entries, and therefore is at most linear in the number of entries.

6. Practical results

To quantify the effects of using the DAG, a small experiment was run. From WordNet [1] the file containing noun data had all of its content tokenized (including words, comments, numbers and separators), and each unique instance was added to an instance of a trie. The number of entries was 268,984, the total number of letters over all the entries (excluding terminators) was 2,636,945. Representing these entries contigously in RAM separated by a null character would require 2,905,929 bytes of storage. In the trie, 1,333,594 nodes was used (that is, approximately half as many nodes as total letters). However the DAG required only 270,488 nodes. In addition the total number of child links for the DAG was only 502,551 giving an average fanout per node less than 2. This means that each node can consist of one letter, one integer for counting children and on average less than 2 pointers. If a non-naive approach is taken to allocating memory for this (and if the alphabet is representable in an 8-bit byte; and thereby the integer for counting children needs only be 1 byte as well), each node can be represented on a 32-bit computer by on average less than $1+1+2*4 = 10$ bytes. This is *less* than the amount of space just to store the strings contigously, and allows efficient lookup. Adding satellite data requires only an additional integer per node for the counting mechanism (making the average node size slightly less than 14 bytes), plus the space used for storing the actual satellite data. No additional per node space is required to support multisets.

While the example above is possibly representative for the strength of the DAG, additional datasets (in particular smaller datasets and non-English datasets) should be examined before any claims can be made of under which circumstances the use of the DAG for dictionary storage can be considered a good choice. A less sloppy extraction of data discarding comments, numbers and separators would as well improve the quality of the experiment.

7. Conclusions

Several different strategies with different tradeoffs for speed and space (including a strategy requiring no extra space) for adding iterators to the trie have been considered. Additionally a compact representation based on an directed acyclic graph has been presented along with some (albeit weak) experimental data that suggests that this can be a space-efficient method of storing static dictionaries, along with per entry satellite data.

Further improvements could consist of adding reverse pointers to the DAG, so that traversal in both direction would become possible. Additionally, attempting to construct the DAG “on-the-fly” without first constructing the trie, by feeding the strings to the algorithm lexicografically sorted, is also worthwhile, as this has potential for consuming much less memory at any point in time.

Additional experiments examining the space use for different datasets could shed some light on the proper circumstances for using the methods proposed.

References

[1] Cognitive Science Laboratory at Princeton University, Wordnet: A lexical database for the English language, Website accessible at `http://wordnet.princeton.edu/` (2006).

[2] J. Bentley, Programming pearls: A spelling checker, *Communications of the ACM* **28** (1985), 456–462.

[3] W. Doeringer, G. Katjoth, and M. Nassehi, Routing on longest-matching prefixes, *IEEE/ACM Transactions on Networking* **4** (1996).

[4] D.E. Knuth, *The Art of Computer Programming, Volume 3: Sorting and Searching*, 2nd Edition, Addison-Wesley (1998).

[5] B. Stroustrup, *The C++ Programmering Language*, special 3rd Edition, Addison-Wesley Professional (2000).

The cost of iterator validity

Nicolai Esbensen

Department of Computing, University of Copenhagen
Universitetsparken 1, DK-2100 Copenhagen East, Denmark
`nies@diku.dk`

Abstract. Iterators are a fundamental part of the C++ standard library. Iterators are objects that point to other objects; call the objects pointed to elements. An iterator can be used to iterate over a collection of elements stored in a data structure, or simply to access the element pointed to. An iterator and the element pointed to live in a close symbiosis; when the element is moved, the iterator may become invalid if it is not updated accordingly. A data structure is said to provide iterator validity if the iterators to its elements are kept valid at all times independent of the element moves.

In this study the cost of the iterator validity is considered. It is shown that for all fundamental data structures (linked lists, resizable arrays, dictionaries, and priority queues) an implementation exists that supports bidirectional iterators, keeps the iterators valid under modifications, executes all iterator operations in $O(1)$ time in the worst case, and uses linear space on the number of elements stored. For random access iterators the problem appears to be more difficult.

Keywords. Data structures, iterators, iterator validity, linked lists, resizable arrays, priority queues.

1. Introduction

Different template libraries available today provide different data structures, with different purposes eg. heap, trees and tables. Most of these structures provide means of traversing the data structure without knowing the underlying details of the structure itself. These methods are usually, if not always, provided through iterator classes. The classes are usually implemented with a specific pattern, that is a collection of methods for traversing the elements. The methods provided signifies the iterator strengths such as random and bidirectional access. Manipulation of iterators resembles the well known pointer operations in the C++ programming language[2], making the traversal familiar to an experienced programmer using the library. Many structures, in particular sequences are ideal for traversal, and does implicitly provide different iterator strengths. Some examples could be linked lists (bidirectional iterator strength), search-trees (random access) and many other structures with the same characteristics. A library programmer of data structures might at some point be aware of the fact that some data structures are not ideal for a linear traversal, and might at this point acknowledge the need of secondary structures maintaining iterator functionality to the user of the structure. The main focus of this article will be various alternatives for such auxiliary data structures and the need of such as well as their maintenance in the manipulation of the primary structure.

In addition to the briefly mentioned iterator strengths, certain requirements or guarantees to the iterator might be added. If a given iterator pointing to an element in the data structure is updated accordingly to any eventual move of the element the iterator is said to be valid. A very simple requirement which quite surprisingly might be very hard to realize in many cases without the implementation of auxiliary structures. This paper explores the use of these auxiliary structures, which will be mentioned as the secondary structure. When the primary structure is mentioned it will be the the structure we wish to provide an iterator for.

2. Causes of invalidation

In order to avoid iterator invalidation, we might want to formalize certain rules of iterator validity. Furthermore we want to isolate the cases which may invalidate iterators.

2.1 Iterator rank maintenance

The rank of the elements in terms of iteration, is defined as the order the elements is traversed that is more formally; for any two elements i, j, in any forward or backward iteration if i precedes j this must be so at all times. In terms of random access and thereby in terms of indices the unequality $i < j$ must be true at all times if ever true. This requirement implicitly leads to the fact that a forward iteration visits the elements in the order of insertion time. Any violations of the rank concept will be referred to as iterator rank invalidation. This requirement is important for all modifying iterations of the data structure, thus violating this requirement might lead to multiple or missing visitations of elements. It should be noted that this kind of ordering contradicts some data-structures specifically sequential structures such as vectors, where insertion at arbitrary indices are allowed. For structures with no natural ordering the principle can be applied, which provides a stronger guarantee than what normally might be expected of iterators.

2.2 Variants of invalidation

The following modifying operations might lead to invalidation of iterators:

Deletion Deletion will in its nature invalidate any iterator pointing to the element deleted. Thus if an iterator is storing elements by reference, the reference is illegal.

Insertion Insertions at arbitrary indices in sequential memory space leads to movement of elements in memory, thus invalidating all iterators referring to successive elements.

Swapping Swapping imposes invalidation by rank and memory address of the elements and should thus be avoided.

3. Running time of iterator operations

The C++ standard[2] sets specific demands to the running time of the iterator operations, which should be constant amortized time. There are no specific rules about the running times of the operations which maintains the iterators, which might affect the running times of the operations on our primary structure. The unwritten rule for the is that no iterator operation should add any surprisingly large overhead [6]. At this point one might ask what is a surprising overhead?First of all, maintaining the secondary data structure should not be more expensive than maintaining our primary data structure, that is for an example, insertion time in our secondary data structure can not be the most costly operation of a given insertion operation. This requirement in itself imposes a very tight bound for many implementations.

4. Invalidation by deletion

In this section, we briefly discuss how deletion should be handled in terms of iterators, because this concept is general for all of the following implementations.

When an element is deleted in our primary structure, thereby no longer existing, iterators referring to this element is invalidated. The common and recommended interface for deletion is this:

```
iterator erase(iterator)
```

The erase operation deletes the element referred to by the iterator, thereafter returning the next element in the sequence. A user of the data structure is thereby aware of the invalidation taking place. Taking validity to extremes; other iterators might exist pointing to the very same element, and also become invalid. The invalidation might affect the program in two ways; the iterator points to overwritten or to deallocated memory. The second case is significant since dereferencing deallocated memory might terminate the program due to segmentation fault caused by illegal memory access. Ensuring correct program behavior in this case can be accomplished fairly simple. Each element can store a possibly empty bucket of iterators currently visiting the element and on deletion set a boolean value indicating that the iterator is no longer valid for all affected iterators. The valid state could be exposed through a method, which the user is advised to check, or accesses to the iterator could throw an exception. This implementation is fairly cheap, since the number of active iterators is expected to be small, if we explicitly require the user to notify the library if the iterator is active or not.

5. A valid implementation-scheme for bidirectional iterators

Depending on the data structure being implemented, the elements might be stored in numerous different ways. Sequential placement in memory is quite common for many data structures, or an internal reference or linkage by pointers, that is scattered placement in memory. The sequential placement in memory does not suffice to provide validity for different compartments, that is the blocks of memory for the stored elements, as elements might be swapped e.g. in heap or tree implementations. The fundamental problem for these kinds of implementations is that the internal structure is implicitly provided by offsets in sequential memory space, which as a consequence leads to physical rearrangement in memory. Pointer based implementations of these structures does not suffer this consequence as structural changes do not likely impose swapping in memory. Thus making the data structure pointer based imposes great benefits for iterator validity, since element swapping can not invalidate the iterator. The problem is solvable even for sequentially allocated structures which will be shown later in this section. For now we will ignore swapping as a source of invalidation and provide a simple implementation of bidirectional iterators.

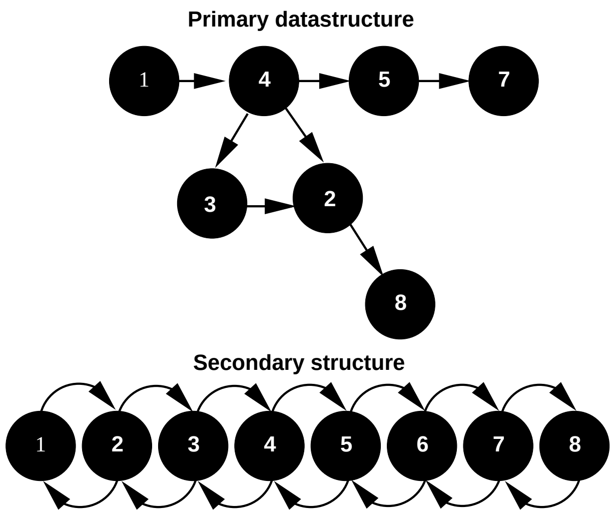

Figure 1. Illustrating the concept of chaining, providing bidirectional iterators. The numbers on the elements are their insertion time. It's clear to see that the iteration through our primary structure is invalid in terms of rank. The elements in the two structures shown are the same, their two relations is shown apart for simplicity of the illustration.

The simplest way of providing bidirectional iterators is to maintain internal pointers between the elements forming a doubly-linked list as illustrated in Figure 1. At initialization of our primary structure we initialize the list as a single circular referenced element, that element being our past-the-end element, our stop criteria in an iteration loop. We furthermore need pointers to the first and the past-the-end element in the list, which we will return when respectively `begin` and `end` invoked. On each insertion we will append the new element at the end of the list updating our last element by replacing its successor, and updating our pointer to the last element. Deleting an element is equally simple, remove the element by updating its predecessor and successor respectively. All operations being constant time, the maintenance of our iterator structure does not imbue the operations on our primary data structure with additional asymptotic overhead. Furthermore the cost in terms of space is obviously linear in the number of elements, the constant being a small factor of the size of two extra pointers. Forward as well as backwards iterators are trivially accomplished using this concept.

6. Providing iterator validity for random access iterators

To provide random access in reasonable times, the data structure providing random access must be efficient. A sequential solution provides fast constant-time accesses, but is as earlier discussed vulnerable in terms of validity. To solve this problem we must ensure a closer relationship between the internal elements and the access structure. To do this the element as well of the index structure, must be more aware of each other. To accomplish this, we will need to have mutual bond between the element and its index. Thus storing a pointer in our index structure to the element as well as a pointer or offset from our element to our index, will enable us to update the element as well as the index should any of those change. This is briefly illustrated in Figure 2. The changes may happen of various causes depending on the two structures.

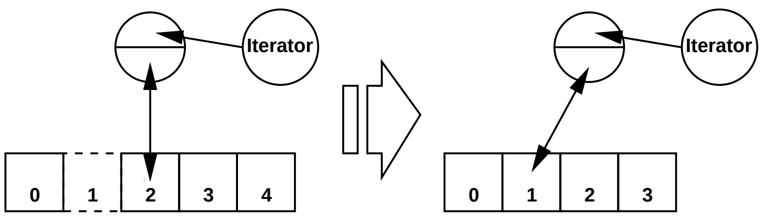

Figure 2. An illustration of how the iterator is maintained. In this example a change in offset. The extra level of indirection allows us to implicitly update any index attached to the element.

As earlier noted the index structure should be chosen carefully, because we do not wish to violate our primary data structure's running times. The maintenance can be quite costly depending on the index implementation chosen. In the following sections we will analyse some common structures and the effect on the running times of primary structure, and furthermore how random access could be provided through them.

6.1 Dynamic resizable arrays

Arrays in its sequential nature are ideal for providing random access functionality, but can in some cases imbue unreasonable high overhead to different operations on our primary structure. Depending on the implementation, several memory copying operations might take place when contracting or expanding the array. In addition, this might also happen when inserting or erasing elements on arbitrary indices, leading to linear time operations on our primary data structure. The most common usage will be pushing elements to the back of the array, which is an $O(1)$ operation, but for structures where deletion is common for example priority queues used for sorting, the maintenance of the index will result in an $O(n^2)$ running time in contrast to the expected $O(n \log n)$ running time of sorting.

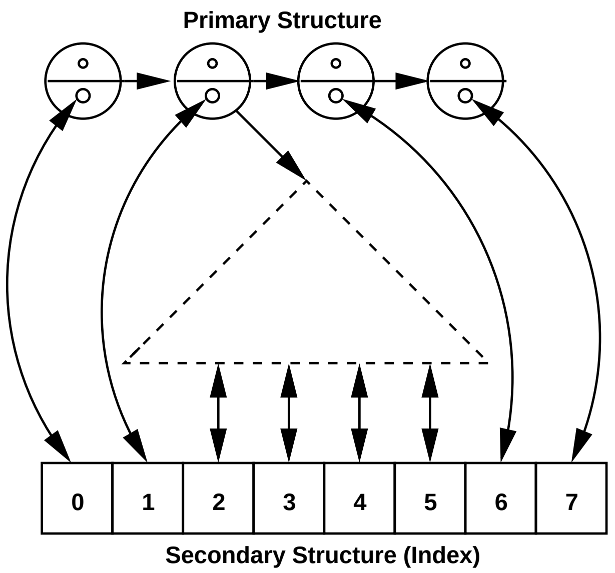

Figure 3. A random access iterator provided an arbitrary structure using a dynamic resizable array. The elements and its index are closely related, which enables us to mutually update the elements and their indices.

The resizable array implementation as proposed by Brodnik et al. reduces deletion and insertion time to a squared factor, and will make a reasonable choice for various purposes, though that is at the cost of an expected higher constant-time operation for indexing the array. This is due to the arithmic structure/system of the the structure. Deletion becomes a lot cheaper since only $O(\sqrt{n})$ elements are invalidated by memory address, although all successive elements will be invalidated by index.

For an overview of the structure, the original article by Brodnik et. al. [3] provides a good overview of the implementation details, the article by Bjarke Buur Mortensen and Jyrki Katajainen [7] points out some complications and errors and might also be of some assistance.

6.2 Using balanced trees as index

The fairly high overhead in delete operations for dynamic resizable arrays, might not be satisfying for delete intensive data structures. If delete is the most common operation, it might be a good idea to consider an implementation of an index using a balanced tree having an upper bound $O(\log n)$ on deletion. The drawback of this solution is that every other operation is in fact of asymptotical logarithmic nature, imposing a higher overhead in the maintenance of the structure. Furthermore, each random access takes logarithmic time, which contradicts the guarantee purposed by the C++ standard. A thorough iteration is though attainable in linear time making each singular-advancing operation[1] constant time using finger search [1], or chaining as previously described.

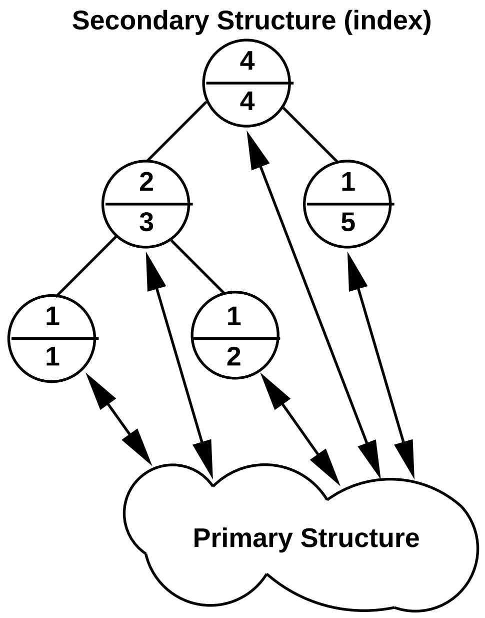

Figure 4. A simple illustration of a random access iterator implemented using a balanced tree. The tree number in nodes top, bottom is reflecting their child count and sequence number respectively.

[1] Eg. `++`, `--`

Using the general idea of random access by mutual reference between the element and its index, in this case the tree node, several time bounds might be improved still maintaining iterator validity. First we might want to discuss how validity is assured in terms of iterator rank. The general idea is to insert the pointers to our elements in our primary structure, using a sequence number reflecting the elements insertion time as our key. This ensures rank validity when traversing the index in- or post-order. The drawback is that the sequence number does not suffice as an index reference in our random access interface, since elements might be deleted making gaps in our sequence. To attain sequential index order, we must either maintain all indices on deletion or replace the search algorithm not using the index, but instead the child count. The first solution might be very costly, having linear worst-case running time. If we modify the search algorithm to use child count instead, we must maintain the child count of all parents when deleting or inserting which does not add additional asymptotic overhead, the number of parents being bounded by $O(\log n)$. Due to the nature of the tree we have that the child count is closely related to the order of insertion, which leads to the following recursive algorithm (Figure 5).

```
node * Search(node * knot, int index){
  if(index == 0)
    return knot;
  if(knot->left->childcount() < index)
    search(knot->right, index - knot->left->childcount())
  else
    search(knot->left, index);
}
```

Figure 5. Code providing sequential index-lookup for balanced binary trees.

The basic idea of the algorithm is that the elements in the left branch is inserted before the elements in the right branch, and we can naturally discard them if their count is less than our index, moving our focus to the right branch. Subtracting the child count can be done since we implicitly have visited the first *child count* elements. If our index becomes zero, we must have visited the first *index* elements and thus has reached our element.

The real drawback to this kind of indexing is that it breaks the possibility of constant time insertion and deletion times. Constant-time insertion and deletion is possible if we store pointers to the nodes corresponding to our elements in our primary structure. Thus deleting an element in our primary structure or inserting one can be done in constant time in our index-tree since we do not need to lookup the tree node because we all ready got a reference to it. The balancing of the tree itself takes constant time, at least for Red-Black trees [4], the total procedure in our index is constant time.

But since we have to update all parents from our node to the root, insertion and deletion becomes logarithmic time operations.

Using a tree as an index might be useful for structures with matching asymptotic running times, and is preferable in structures where deletion is common, thus logarithmic running time on deletion is the presented. If random access is very common this structure is not preferable.

7. Fragmentation of the primary structure and iterator sequence

As a last point in this paper, fragmentation in the iterator sequence as well as in our primary structure should be considered. The general rule for the presented implementations is that elements are ordered by their time of insertion. This principle has advantages for the time used for a thorough iteration over the elements of the structures, since elements are allocated in the same order they are visited. This means, depending on the memory allocator, that we are making effective use of burst reads in memory since the ordering in memory is the same as the order of our visitations.

All though this kind of sequential allocation gives great benefits for iteration, it might be a complication for the actual running times for the primary structure. As an example a hash table could be considered. The placement in the table, should ideally be as random as possible which contradicts our sequential allocation. For fast lookups in the table, the elements in the buckets of the individual entries in the hash table should be close in memory. Thus locating elements in memory with respect to the individual buckets, might result in slow iterator operations since the memory accesses might be very random.

For fast iterations in the table, we might want to iterate through the table ordered by the the buckets instead of the time of insertion, thereby taking advantage of the spatial locality of the elements in the buckets. This will violate the iterator rank principle, when new elements are inserted, deleted or rehashed on contraction or expandation of the table, but iteration will be much faster.

Generally for obtaining fast iterations, the ordering of elements should respect the location of elements in memory. From this observation it can be conclude that maintaining the iterator rank might be more costly than just maintaining the sequential order of insertion time. The guarantee it provides is much stronger than what might be necessary for simple non-mutating iterations. All operations for the random access structures, can be reduced to asymptotic constant time if we ignore the ordering of the elements, and simply on deletion replace the gap with the last element in the iterator sequence. One should note that this in addition to the iterator rank invalidation, can over time severely fragment the iterator sequence. The mentioned procedure is only legit if the primary structure has no natural ordering, for example this operation is not applicable for vectors since the ordering is explicitly specified by the user.

8. Conclusion

In this paper several implementations for of iterators of different strengths were presented. The bidirectional iterator being the less costly of all, and validity is easily attainable for all data structures by means of chaining. The presented implementation imposes no asymptotic overhead, all operations being constant time, and the ease of the implementation makes it recommendable solution to provide different iterator strength, bidirectional being the strongest.

Random access is complicated to realize efficiently in general, and ensuring validity is asymptotically costly. The most costly guarantee to attain is undoubtly iterator rank validity, which is common to be violated on deletion of elements in the primary structure. Asymptotically we have seen three different implementations ensuring this guarantee in $O(n)$, $O(\sqrt{n})$ and $O(\log n)$ for doubly resizable arrays, a pile implementation and a balanced tree, respectively. The constant involved in updating successive elements being invalidated on deletion is in all random access implementations reduced to dereferencing a pointer, and updating the corresponding element with the new index/address which is in worst case requires two memory accesses. Figure 6 shows the impact of the extra level of indirection, which is reasonably small.

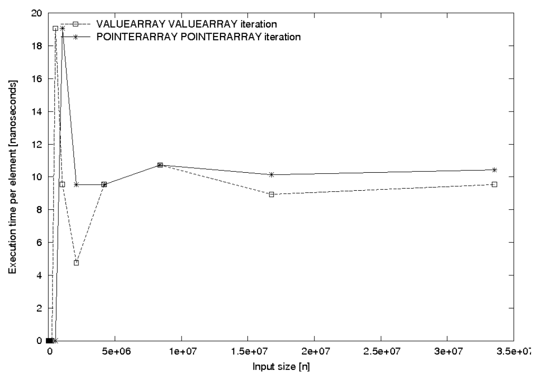

Figure 6. A graph which shows the impact of the extra level of indirection, when iterating trough our index. It can be seen that the extra level of indirection is not imposing an unreasonable large overhead. The benchmark was run on an Intel Pentium 4 3.0 GHz processor with 2 Gigabytes of RAM in an Unix environment.

The choice of how random access should be provided, should be considered with respect to the structure being implemented. The presented solutions has cons as well as pros, and valid random access might be to costly for many structures to be reasonable to provide. Alternatively one could consider, giving the user the option of specifying the iterator strength as a template parameter, thereby letting the user decide whether or not the overhead is acceptable.

Acknowledgements

I would like to thank Jyrki Katajainen, for guidance writing this paper. Furthermore he should be credited for many of the presented ideas, which is originated from his presentation slides [6] on the subject with the same title as this paper. Furthermore i would like to thank the other students on the course, who have contributed with constructive reviews of the draft of this paper.

References

[1] G. E. Blelloch et al., Space-efficient finger search on degree-balanced search trees (2002).
[2] British Standards Institute, *The C++ Standard: Incorporating Technical Corrigendum 1*, 2nd Edition, John Wiley and Sons, Ltd. (2003).
[3] A. Brodnik, S. Carlsson, E. D. Demaine, J. I. Munro, and R. Sedgewick, Resizable arrays in optimal time and space (1999).
[4] T. H. Cormen et al., *Introduction to algorithms*, 2nd Edition, The MIT Press (2001).
[5] R. Hinze and R. Paterson, Finger trees: a simple general-pupose data structure. (2004).
[6] J. Katajainen, Presentation - the cost of iterator validity (2004).
[7] J. Katajainen and B. B. Mortensen, Experiences with the design and implementation of space-efficient deques (2001).
[8] M. D. Kristensen, Vector implementation for the cph stl (2004).

A tool kit for optimizing recursive functions

Rune Møllegård Madsen

Department of Computing, University of Copenhagen
Universitetsparken 1, DK-2100 Copenhagen East, Denmark
`downey@diku.dk`

Abstract. A study into the possibilities of optimizing recursive functions. Some programs are easier to implement using recursive functions, unfortunately they consume much more memory than the iterative version of the same algorithm. I have tried to create a tool kit for the C++ language that will make it possible to write your algorithm in the recursive form, but will produce an iterative version that will run at about the same speed or a bit faster, consuming less memory. Template programming and macro processing are used to create an optimized version of a recursive function.

Keywords. Recursive, optimization, tail recursion, macros, templates

1. Introduction

Many algorithms are more simple and intuitive to implement in their recursive form, than in their corresponding iterative form. This paper is about how to run your recursive implementation as fast as possible, taking advantage of macros and templates in C++.

It is investigated how a recursive implementation is executed, and what measures of rewriting it would require to make an iterative implementation. An automatic conversion by a tool kit is then looked upon and it is concluded that it is too difficult to create using templates. An unrolling of the recursive implementation was tried, by using the `inline` keyword in C++ and templates. The `GCC` C++ compiler optimizer already does some automatic in-lining, which makes the unrolling by templates redundant. A tool kit is created to make it easier to implement a tail recursive algorithm into an generic iterative implementation using macros and templates. I conclude that, a method was found to imitate the iterative form of a tail recursive algorithm. There was not found a method that made the general recursive algorithm run any faster.

2. Background

A recursive function is a function that as an important part of its execution, calls itself. To stop the recursion, a terminating condition is used and the recursion will run until this condition is true. When a terminating condition is true, that recursion branch will end, but other recursive branches can be in use and all branches has to end for the entire function to end.

Another recursion are indirect calls, where for example a statement in `a()` calls `b()` and a statement in `b()` calls `a()`. Indirect calls will not be discussed further in this study.

A common use of recursive algorithms is in divide and conquer algorithm technique, where problems are divided into subproblems, solved, and then combined into solutions. Some other use is to solve simple mathematical functions like `pow` or `Fibonacci`. The simplest and most direct method to implement a recursive algorithm is to use recursive functions. When you execute a recursion, the callstack will grow for each new recursive call. This puts a limit on the size of data that you can work on before running out of memory. Also, it produces an overhead to maintain the callstack and load/save a stackframe for each function call and we would like to avoid this.

Templates are an addition to the original C++, that makes it possible to write generic classes and functions. Using the features of templates one can also use metaprogramming and this way compile specific parts of one's code at compile time, discover errors at compile time, as well as do some optimizations using templates. Templates are explained in more detail by Vandevoorde and Josuttis [2].

Macro processing is done at compile time as templates and makes it possible to define functions that will be replaced at compile time. Several compilers exist that supports templates and macro processing and are all capable of compiling C++ sourcecode into executable binaries. I have decided to focus purely on `GCC` as it is one of the most used and advanced compilers out there. More info on macro processing and the `GCC` compiler can be found in the online documentation for `GCC` [1].

3. Imitating recursion as iterative

The following six steps, will be able to imitate **any** recursive function as an iterative function. It simply copies the functionality of a callstack from a compiler which is also our guarantee, that the conversion is correct.

1. Initialize a stack, that will hold local variables, call values, return values, and jump labels for each recursive call.
2. Encapsulate the body of the function in a while loop, that will loop until the stack is empty.
3. Divide the body into the following steps, "from the beginning to a recursive call", "after a recursive call to the end". In case of more than one recursive call, you just insert a section more between each of the calls. To divide the body into sections and be able to jump into a section, I suggest to use a switch statement.
4. Replace all recursive calls to statements that will save the local values,

call values, and the jump label, so that when the loop (corresponding to the recursive call) ends, it will continue to execute from the next section.

5. At the end of the final section in the switch statement, save the return value and jump to the correct section by looping and switching to the jump label.
6. When the while loop ends, pop the result of the recursion from the stack.

As mentioned in Section 2 there are different forms of recursive algorithms: the ones that execute statements after the return of a recursive call; divide and conquer algorithms are in this group, and others where the recursion is part of the final statement in the function. This kind of recursion is called *tail recursion* and is simpler to rewrite into an iterative form. An example of this function is the classic factorial recursion.

All tail recursion algorithm can be rewritten to use accumulators, and in our study we need an accumulator, because it contains the current calculated result, which means we don't have to save any other data when we return from a recursive call.

```
1: int factorial_rec(int x, int acc=1) {
2:        if (x <= 1)
3:                return acc;
4:        return factorial_rec(x-1,acc*x);
5: }
```

Converting a tail recursive function to an iterative function is done by doing the following three steps.

1. Encapsulate the body of the function in a while loop.
2. Remove the accumulator `acc` from the parameter list and initialize it instead before the while loop.
3. replace the tail call at line 4, with a statement for each parameter, that will update the values. The result can be seen in `factorial_iterative` line 6 and 7.

```
1: int factorial_iterative(int x) {
2:         int acc=1;
3:         while (true) {
4:                 if (x <= 1)
5:                         return acc;
6:                 acc= acc*x;
7:                 x=x-1;
8:         }
9: }
```

To see that this conversion is correct we only need to understand that the executions of the two functions, executes the same expressions and assignments, and is limited by the same terminating expression. It is quite clear, that this is the case.

4. Optimization

The goal of this section is to address two different ways to optimize recursive functions by giving developers a set of tools to use. Two different ideas have been targeted for inclusion in this tool kit. The methods will be explained separately in the next two subsections.

4.1 Tail recursion

When converting a tail recursive function into an iterative one, we do not need to save the variables from each recursive step, we only need to take care of the result of each iteration. Because the steps are simple we can use macros to convert a tail recursive function. It is important that the use of macros will resemble the creation of a recursive function; otherwise they will not be of any use. To make sure that any tail recursion can be handled we use templates that makes it possible to specify template arguments when calling the function. The type of the accumulator can be specified, which will also be the return type of the function. The type of the other arguments will be automatically deducted from the type of the parameter when calling the function, and if wanted, it can also be specified.

We take the three steps from Section 3 and rewrite them into three steps that will build an iterative function resembling tail recursion.

1. Create the function header `TALE_RECURSION_INIT(name, acc_init, ...)`, specify function name, the initialization value of an accumulator, and the name of the internal variables. The initialization of the accumulator could be replaced with `ACCUMULATOR(1)`, using the default initialization of the type specified. This would be very useful, if the accumulator for example was a user defined object.
2. Write your terminating condition using the name of your internal variable and then to get a hold of the tail recursion result, use the value `TAIL_RECURSION_ACC`.
3. (Optional) You are free to write any code before and after step 2.
4. At last, create the recursive call `TAIL_RECURSION_END( update statement, accumulator expression )`, specifying how to update the internal variables as the first parameter. If your terminating condition is depending on variables never updated, your function may never end. The second parameter updates the accumulator, it could of course be calculated in the body and then just be given as a single variable, but our example is so simple so we specify it with an expression.

These are the macros available in my tool kit:

```
#define TAIL_RECURSION_INIT(name, acc_init, ...) \
        template<typename ACCUMULATOR, typename ARG_TYPE> \
        ACCUMULATOR name(__VA_ARGS__) { \
        ACCUMULATOR acc=acc_init; \
        while (true) { \

#define TAIL_RECURSION_ACC acc
#define TAIL_RECURSION_END(param1, paramX) \
TAIL_RECURSION_ACC = paramX; \
        param1; \
        } }\

#define TAIL_RECURSION_ONELINE(name, acc_init, cond, \
                                dec ,accumulate, ...) \
        TAIL_RECURSION_INIT(name, acc_init, __VA_ARGS__); \
        if (cond) \
                return TAIL_RECURSION_ACC; \
        TAIL_RECURSION_END( dec, accumulate );
```

When using the macros we can create an iterative form of our `factorial_rec` mentioned in Section 3.

```
TAIL_RECURSION_INIT(factorial_tail, 1, ARG_TYPE x);
        if (x <= 1)
                return TAIL_RECURSION_ACC;
TAIL_RECURSION_END( x= x-1, TAIL_RECURSION_ACC * x);
```

Actually the function created by our macros is generic whereas our original function is not. The original could of course be made generic, but my point is that we achieve this in our function without making the process of writing the function more complex.

As a final macro, `TAIL_RECURSION_ONELINE` is created to be able to quickly create several recursive functions, an example is listed here. Be aware that readability takes a hit when using this oneline macro compared to using the others.

```
TAIL_RECURSION_ONELINE(factorial_tail_oneline, 1,
                       x <= 1,
                       x= x-1,
                       TAIL_RECURSION_ACC * x, ARG_TYPE x);
```

No name collisions should occur when using the tool kit, as long as distinct names are used for the generated functions. It is expected that functions generated with these macros will execute with less instructions than the equivalent tail recursive function, since the maintenance of a callstack is avoided. This is the most promising optimization of the two.

4.2 Recursion and inline functions

Template metaprogramming is used because it lets us create recursive functions at compile time. We can use this metaprogramming to unfold the recursive calls in a function, in-lining to N level. When a function is called for example `Recursive<10>::pow(5,14)` it will then call the in-lined version the first 9 recursions. From the 10 level and so on we have $N = 1$, which stops in-lining.

```
template <int N>
class Recursive {
        public:
        inline static double pow(double x, int y) {
                if (y <= 1)
                        return x;
                return (x*Recursive<N-1>::pow(x,y-1));
        }

};

class Recursive<1> {
        public:
        static double pow(double x, int y) {
                if (y <= 1)
                        return x;
                return (x*pow(x,y-1));
        }
};
```

The idea for this implementation was that one cannot simply create an in-lined recursive function because this would make the compiler go into an infinite loop. Meanwhile I have discovered that this is flawed, because the GCC compiler actually does a splendid job when in-lining an in-lined recursive function.

5. Conclusions

Nowadays compilers are so powerful that it is difficult to find areas to optimize, without just being able to reach same run-times by compiling with $-O3$ or comparable optimization parameters. `GCC` as an example takes among many others the following optimization arguments `-finline-functions -finline-limit=n -foptimize-sibling-calls`. If all the compile options are used and n (n sets the limit for the size of in-lined functions, all functions below this size in bytes will be in-lined) is set high, then GCC will create larger executables in-lining almost every function making it execute a lot faster. The idea behind optimizing by doing in-lining is not that relevant as it seems, because `GCC` is so good at it.

This leaves us with our new macros for writing tail recursive functions that are actually iterative. The tool kit is short, simple and it will likely produce more effective code than the corresponding tail recursive version.

References

[1] GCC team, GCC online documentation, Website accessible at `http://gcc.gnu.org/onlinedocs/` (2006).

[2] D. Vandevoorde and N. M. Josuttis, *C++ Templates The Complete Guide*, 1st Edition, Addison-Wesley, New York (2002).

Author index