Modern Assembly Language Programming with the ARM processor

Chapter 6: Abstract Data Types

Contents

ADT

An Abstract Data Type (ADT) is composed of:

- data and
- the operations that work on that data.

Separate the *interface* from the *implementation*!

Object Oriented Programming (OOP) is a related concept with similar goals.

- C provides support for ADTs and allows the implementation to be done in assembly.
- C++ provides support for objects and allows the implementation to be done in assembly (but you need to know a bit more about the compiler).

Information Hiding

- Hide the physical storage layout for data.
- If the data structure changes, it only affects a small subset of the total program
    - Unit Testing
    - Rapid Prototyping
    - Standardization
    - Debugging
    - Code Reuse
    - Independent Development
    - Larger Projects
    - Higher Quality

ADT Interface in C

```c
#ifndef IMAGE_H
#define IMAGE_H
#include <stdio.h>

typedef unsigned char pval;

struct imageStruct;

typedef struct imageStruct Image;

Image *allocateImage();
void freeImage(Image *image);

int readImage(FILE *f, Image *image);
int writeImage(FILE *f, Image *image);

int setPixelRGB(Image *image,int row,int col,pval r,pval g,pval b);
int setPixelGray(Image *image, int row, int col, pval v);

pixel getPixelRGB(Image *image, int row, int col);
pval getPixelGray(Image *image, int row, int col);

#endif
```

ADT Private Declarations

A separate header, only available to the Image implementation files:

```
#ifndef IMAGE_PRIVATE_H
#define IMAGE_PRIVATE_H
#include <image.h>

typedef struct {
  pval r,g,b;
} Pixel;

struct imageStruct;
  int rows;      // number of rows in the image
  int cols;      // number of columns in the image
  Pixel *pixels; // array of pixel data
  };
#endif
```

ADT Private Declarations

imagedefs.S

```
@@@ Definitions for pixel and image data structures

       @@ pixel
        .equ    p_red,   0  @ offset to red value
        .equ    p_green, 1  @ offset to green value
        .equ    p_blue,  2  @ offset to blue value
        .equ    p_size,  3  @ size of the pixel data structure

        @@ image
        .equ    i_rows,  0  @ offset to number of rows
        .equ    i_cols,  4  @ offset to number of columns
        .equ    i_pixels,8  @ offset to pointer to image data
        .equ    i_size, 12  @ size of the image data structure
```

ADT Implementation in Assembly

```
        .include "imagedefs.S"  @ include data layout
        .text
/* Implementation of
int setPixelRGB(Image *image,int row,int col,pval r,pval g,pval b);
*/
        .global setPixelRGB
setPixelRGB:
        stmfd   sp!,{r4,r5}      @ We will need some registers
        @@ Calculate address of desired pixel
        ldr     r4,[r0,#i_cols] @ get # colummns in image
        mla     r4,r4,r1,r2      @ calculate offset to desired row
                                 @ and add offset to desired column
        mov     r5,#p_size       @ load size of a pixel
        mul     r4,r4,r5         @ get offset to desired pixel in bytes
        ldr     r5,[r0,#i_pixels]@ get pointer to the pixel array
        add     r0,r4,r5         @ add offset to desired pixel
        @@ Set the rgb fields in the pixel
        ldr     r4,[sp,#8]       @ load g from stack
        ldr     r5,[sp,#12]      @ load b from stack
        strb    r3,[r0,#p_red]   @ store r,g,b in pixel struct
        strb    r4,[r0,#p_blue]
        strb    r5,[r0,#p_green]
        ldmfd   sp!,{r4,r5}
        mov     pc,lr
```

Faster ADT Implementation in Assembly

We know that a pixel is three bytes, so replace the multiply with a shift/add operation.

```
        .include "imagedefs.S"  @ include data layout
        .text
/* Implementation of
int setPixelRGB(Image *image,int row,int col,pval r,pval g,pval b);
*/
        .global setPixelRGB
setPixelRGB:
        stmfd   sp!,{r4,r5}      @ We will need some registers
        @@ Calculate address of desired pixel
        ldr     r4,[r0,#i_cols] @ get # colummns in image
        mla     r4,r4,r1,r2     @ calculate offset to desired row
                                @ and add offset to desired column
        add     r4,r4,r4,lsl #1 @ multiply by 3
        ldr     r5,[r0,#i_pixels]@ get pointer to the pixel array
        add     r0,r4,r5        @ add offset to desired pixel
        @@ Set the rgb fields in the pixel
        ldr     r4,[sp,#8]      @ load g from stack
        ldr     r5,[sp,#12]     @ load b from stack
        strb    r3,[r0,#p_red]  @ store r,g,b in pixel struct
        strb    r4,[r0,#p_blue]
        strb    r5,[r0,#p_green]
        ldmfd   sp!,{r4,r5}
        mov     pc,lr
```

Word Frequency Counts

Counting the frequency of words in written text has several uses.

- Digital forensics: provide evidence as to the author of written communications.
- Document Classification: suggest similar books or help with sorting documents for storage and access.

Many ways to implement this ADT!

- Linked List
- Binary Tree
- Trie
- ...

Wordlist Interface in C

```c
#ifndef LIST_H
#define LIST_H

/* Define an opaque type, named wordlist                   */
typedef struct wlist wordlist;

/* wl_alloc allocates and initializes a new word list.    */
wordlist* wl_alloc();

/* wl_free frees all the storage used by a wordlist       */
void wl_free(wordlist* wl);

/* wl_increment adds one to the count of the given word.  */
/* If the word is not in the list, then it is added with  */
/* a count of one.                                         */
void wl_increment(wordlist *list, char *word);

/* wl_print prints a table showing the number             */
/* of occurrences for each word, followed by the word.    */
void wl_print(wordlist *list);

/* wl_print_numerical prints a table showing the number   */
/* of occurrences for each word, followed by the word,    */
/* sorted in reverse order of occurence.                   */
void wl_print_numerical(wordlist *list);

#endif
```

Wordlist Private Declarations

C Data structures:

```c
/* The wordlistnode type is a linked list of words and     */
/* the number of times each word has occurred.             */
typedef struct wlist_node{
  char *word;
  int count;
  struct wlist_node *next;
}wordlistnode;

/* The wordlist type holds a pointer to the linked list    */
/* and keeps track of the number of words in the list      */
typedef struct wlist{
  int nwords;
  wordlistnode *head;
}wordlist;
```

Assembly definition:

```
@@@ Definitions for the wordlistnode type
        .equ     wln_word,0     @ word field
        .equ     wln_count,4    @ count filed
        .equ     wln_next,8     @ next field
        .equ     wln_size,12    @ sizeof(wordlistnode)
@@@ Definitions for the wordlist type
        .equ     wl_nwords,0    @ number of words in list
        .equ     wl_head,4      @ head of linked list
        .equ     wl_size,8      @ sizeof(wordlist)
```

Problems with Wordlist Implementation

Linked List:

- Inserting or updating an item in the list requires $\frac{n}{2}$ comparisons on average to find the insertion point. It is an $O(n)$ algorithm. It is very slow for large text inputs.
- Re-sorting the list by frequency is also very slow.

Binary Tree?

- Inserting or updating an item in the tree requires $\log_2 n$ comparisons on average. It is an $O(\log n)$ algorithm.
- It is much faster than a linked list.

Worst case, there are about $1,000,000$ words in the English language. $500,000$ comparisons for linked list versus 19.9 comparisons for tree. Speedup of ≈ 25125.

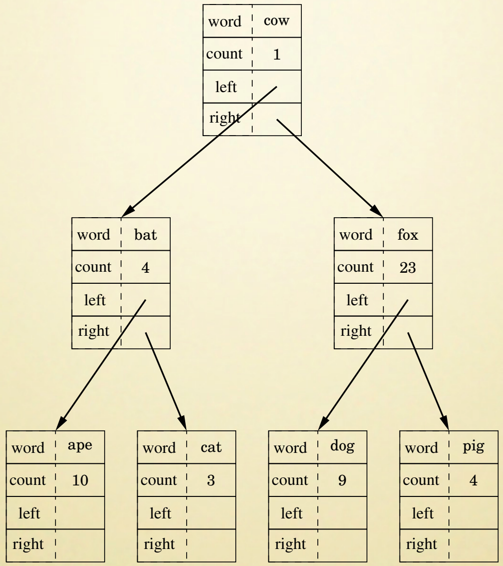
Problems with Wordlist Tree-based Implementation

Re-sorting the list by frequency could still be slow.

Solution: Add an index, then sort the index with quicksort!

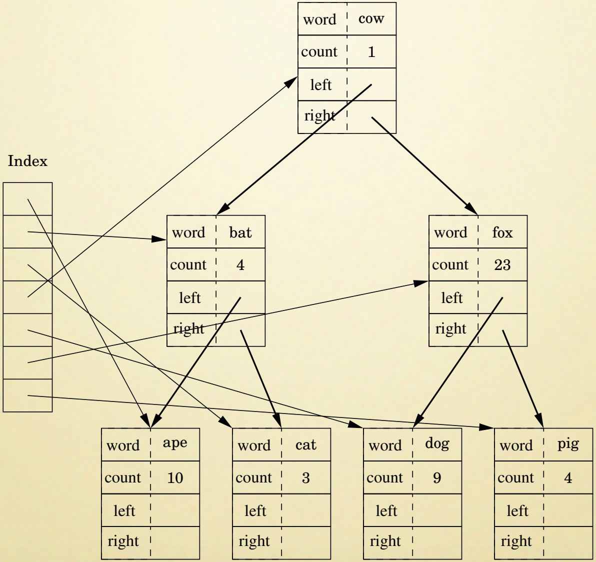
The index is just an array of pointers; one for each node in the tree.

Binary Tree of Word Frequencies



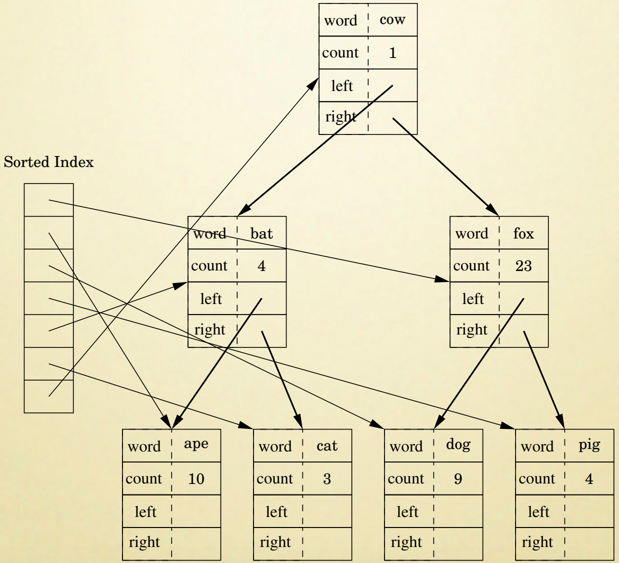
Binary Tree of Word Frequencies with Index

The index was initialized using an in-order traversal of the tree.



Binary Tree of Word Frequencies with Index

The index was sorted using a quicksort.



Wordlist Private Declarations: Tree Version

These changes are not visible to any program using this ADT.

C Data structures:

```c
/* The wordlistnode type is a binary tree of words and    */
/* the number of times each word has occurred.            */
typedef struct wlist_node{
  char *word;
  int count;
  struct wlist_node *left, *right;
  int height;
}wordlistnode;
```

Assembly definition:

```
@@@ Definitions for the wordlistnode type
        .equ     wln_word,0     @ word field
        .equ     wln_count,4    @ count field
        .equ     wln_left,8     @ left child pointer
        .equ     wln_left,12    @ right child pointer
        .equ     wln_height,16  @ height of this node
        .equ     wln_size,20    @ sizeof(wordlistnode)
```

Printing in Order of Frequency part 1

```
@@@ -----------------------------------------------------------
@@@ wl_print_numerical prints a table showing the number
@@@ of occurrences for each word, followed by the word,
@@@ sorted in reverse order of occurence.
        .global wl_print_numerical
wl_print_numerical:
        stmfd   sp!,{r4-r6,lr}  @ save registers
        mov     r4,r0           @ copy original pointer
        ldr     r5,[r0,#wl_nwords]@ load nwords
        lsl     r0,r5,#2        @ multiply by four (size of pointer)
        bl      malloc          @ allocate storage for the index
        cmp     r0,#0           @ check return value
        bne     malloc_ok
        ldr     r0,=failstr     @ load pointer to string
        bl      printf
        mov     r0,#1
        bl      exit            @ exit(1)
malloc_ok:
```

Printing in Order of Frequency part 2

```
malloc_ok:
        mov     r6,r0           @ save pointer to array
        ldr     r1,[r4,#wl_head]@ get pointer to tree
        bl      getptrs         @ fill in the pointers
        mov     r0,r6           @ get pointer to array
        add     r1,r0,r5,lsl #2 @ get pointer to end of array
        sub     r1,r1,#4
        bl      wl_quicksort    @ re-sort the array of pointers
        @@ Print the word frequency list.
        mov     r4,#0           @ do a for loop
loop:   cmp     r4,r5
        bge     done
        ldr     r0,=fmtstr
        ldr     r3,[r6,r4,lsl #2] @ get next pointer
        add     r4,r4,#1
        ldr     r1,[r3,#wln_count]@load count
        ldr     r2,[r3,#wln_word] @load ptr to word
        bl      printf
        b       loop
done:   ldmfd   sp!,{r4-r6,pc}  @ restore & return
```

Initializing the Array of Pointers (the index) part 2

```
@@@ -----------------------------------------------------------
@@@ wordlistnode **getptrs(wordlistnode *ptrs[],wordlistnode *node)
@@@ this function recursively traverses the tree, filling in the
@@@ array of pointers.
@@@ r0 is incremented as each pointer is stored, so it returns
@@@ a pointer to the next pointer in the array that needs to
@@@ be set.
getptrs:
        cmp     r1,#NULL            @ if node == NULL
        moveq   pc,lr               @ return immediately
        stmfd   sp!,{r4,lr}
        mov     r4,r1               @ save address of current node
        ldr     r1,[r4,#wln_left]   @ get ptr to left child
        bl      getptrs             @ process left child
        str     r4,[r0],#4          @ Store address of current node
        ldr     r1,[r4,#wln_right]@ get ptr to right child
        bl      getptrs             @ process right child
        ldmfd   sp!,{r4,pc}
```

Quicksort on Array of Pointers (the index) in C

```c
wl_quicksort(wordlistnode **left,wordlistnode **right)
{
  wordlistnode **first=left, **last=right;
  wordlistnode *tmp;
  int pivot;
  if(left < right)
    {
      pivot=(*left)->count;
      do{
        while((left <= right) && ((*left)->count > pivot))
          left++;
        while((left <= right) && ((*right)->count < pivot))
          right--;
        if( left <= right )
          {
            tmp = *left;
            *left = *right;
            *right = tmp;
            left++;
            right--;
          }
      }while(left<=right);
      wl_quicksort(first,right);
      wl_quicksort(left,last);
    }
}
```

Quicksort on Array of Pointers (the index) part 1

```
@@@ -----------------------------------------------------------
@@@ function wl_quicksort(wln **left,wln **right) quicksorts
@@@ the array of pointers in order of the word counts
wl_quicksort:
        cmp     r0,r1
        movge   pc,lr           @ return if length<=1
        stmfd   sp!,{r4-r7,lr}
        ldr     r12,[r0]        @ use count of first item as
        ldr     r12,[r12,#wln_count] @ pivot value in r12
        mov     r4,r0           @ copy current left
        mov     r5,r1           @ copy current right
        mov     r6,r0           @ original left(first)
        mov     r7,r1           @ original right(last)
loopa:  cmp     r4,r5           @ while ((left <= right) &&
        bgt     finale
        ldr     r0,[r4]         @ ((*left)->count > pivot))
        ldr     r1,[r0,#wln_count]
        cmp     r1,r12
        ble     loopb
        add     r4,r4,#4        @ increment left
        b       loopa
```

Quicksort on Array of Pointers (the index) part 2

```
loopb:  cmp     r4,r5           @ while left <= right &&
        bgt     finale
        ldr     r2,[r5]         @ (*right)->count < pivot
        ldr     r3,[r2,#wln_count]
        cmp     r3,r12
        bge     cmp
        sub     r5,r5,#4        @ decrement right
        b       loopb
cmp:    cmp     r4,r5           @ if( left <= right )
        bgt     finale
        str     r0,[r5],#-4     @ swap pointers and
        str     r2,[r4],#4      @ change indices
        b       loopa
finale: mov     r0,r6           @ quicksort array from
        mov     r1,r5           @ first to current right
        bl      wl_quicksort
        mov     r0,r4           @ quicksort array from
        mov     r1,r7           @ current left to last)
        bl      wl_quicksort
        ldmfd   sp!,{r4-r7,pc}
```

Therac-25

The Therac-25 was a device designed for radiation treatment of cancer.

Although this machine was built with the goal of saving lives, between 1985 and 1986, three deaths and other injuries were attributed to the hardware and software design of this machine.

Therac-25

The worst problems in the design and engineering of the machine were:

- The code was not subjected to independent review.
- The software design was not considered during the assessment of how the machine could fail or malfunction.
- The operator could ignore malfunctions and cause the machine to proceed with treatment.
- The hardware and software were designed separately and not tested as a complete system until the unit was assembled at the hospitals where it was to be used.
- The design of the earlier Therac-6 and Therac-20 machines included hardware interlocks which would ensure that the X-ray mode could not be activated unless the tungsten radiation shield was in place. The hardware interlock was replaced with a software interlock in the Therac-25.
- Errors were displayed as numeric codes, and there was no indication of the severity of the error condition.

Therac-25

Although the code was written in assembly language, that fact was not cited as an important factor.

The fundamental problems were

- poor software design,
- over-confidence, and
- re-use of code in an application for which it was not initially designed.

A proper design using established software design principles, including structured programming and abstract data types would almost certainly have avoided these fatalities.