UNIVERSITY OF COPENHAGEN
Department of Computer Science
Robert Glück glueck@diku.dk
Jost Berthold berthold@diku.dk

February 10, 2012

G-Exercise 1 for "Operating System and Multiprogramming", 2012

Hand in your solution by uploading files in Absalon before Feb.21

Exercise Structure for the Course

As in previous years, the course "Operating Systems and Multiprogramming" (Styresysteme og Multiprogrammering) includes a number of G-exercises (Godkendelsesopgave, approval exercises) that are *published on Fridays* and have to be *handed in until Monday (at midnight, 23:59).* G-exercises should be solved in groups of two students. Each G-exercise will be graded with either 0, $\frac{1}{2}$ or 1 point; and you have to obtain a total of at least $3\frac{1}{2}$ points. As there will be five exercises in the 2012 course, at least four exercises need to be handed in to achieve this. For the first four exercises, there will be the possibility of resubmitting an improved solution one week later.
Solutions to G-exercises are in general handed in by uploading an archive file (zip or tar.gz) to Absalon. When you upload your solution to Absalon, please use the last names of each group member in the file name and the exercise number as the file name, like so:
`lastname1_lastname2_G1.zip` or `lastname1_lastname2_G1.tar.gz`.
Use simple txt format or pdf for portability when handing in text, and document the code by comments. When you write C code, it is required to also write a Makefile which enables the teaching assistants to compile your code by a simple "make" invocation.
Please also read the document *Krav til G-opgaver* (in Danish) provided on Absalon. It explains the idea of the exercises and general requirements for code that you hand in.

About this Exercise

Topics

The first exercise is about pointer programming and explicit memory allocation in C, and also an exercise in C programming in general. You are asked to implement different data structures which use pointers to link the contained elements to each other, namely a simple linked list and a special kind of tree. Using the list, one can implement a stack and a queue. The tree serves as an implementation of a flexible array that can grow and shrink on both ends. Container structures like this are widely used in system programming. Explicit memory access and pointers are also typical for low-level systems code.

What to hand in

Please hand in your solution by uploading a single archive file (zip or tar.gz format) which contains:

1. A text or pdf file containing explanations to the code and answers to the questions.
2. A directory with C code for task 1: `list.h`, `list.c`, `stack.h`, `stack.c`,and a Makefile.
3. A directory with C code for task 2: `braunseq.h`, `braunseq.c`, a Makefile, and a test program.

As a minimum, the C code you hand in should compile without errors. For best results, you should also eliminate all warnings issued by the compiler when flag `-Wall` is used.

Tasks

1. **Linked Lists**

   Linked lists are useful data types for system programming in many areas. They can for example be used as a simple queue implementation for scheduling. In this exercise, we start with a simple list structure where each node contains some data and a pointer to its successor. Any data type could be the content of a node; we define the `Data` type to be `void*` for task 1.c).

   (a) The shown interface file `list.h` is provided on Absalon. Implement the declared functions in a file `list.c`.

   i. Implement `length` and `head`.

   ii. Implement `append` and `prepend`. Be careful to allocate memory as required.

   iii. After you have implemented `append` and `prepend`, it will be clear why they use a parameter of type `Listnode** start` – write a short explanation.

   list.h

```
typedef void* Data;
typedef struct listnode {
  Data content;
  struct listnode* next;
} Listnode;

/* add an element at the end of the list */
void append(Listnode **start, Data elem);

/* add an element at the front */
void prepend(Listnode **start, Data elem);

int length(Listnode *start); // length

/* return first element (if not empty) */
Data head(Listnode *start);
```

   (b) The procedure `remv` shown on the right uses a match function (function pointer) and removes the first matching element from the list (if one is found). Using the address of the `next` pointer in a list node (a pointer to a pointer) saves a temporary variable. However, there is a subtle bug in the procedure that makes it not work in some cases.

   i. Describe a case where the procedure does not work correctly.

   ii. Correct the remove function and add it to the list interface and implementation.

   Add to list.c

```
void remv(Listnode **start, int (*match)(Data))
{
  Listnode *curr, **last;
  if (*start == NULL) return;
  last = start;
  curr = *start;
  while (!(match(curr->content)) && curr!=NULL ) {
      last = &(curr->next);
      curr = curr->next;
  }
  if (curr == NULL)
      return; /* not found, do nothing */

  *last = curr->next;  /* found, remove and free */
  free(curr);
}
```

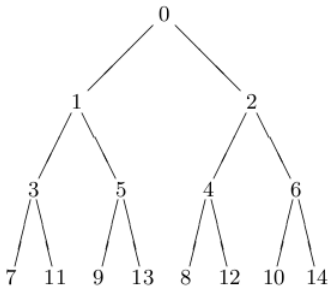
   (c) A simple linked list is a convenient way to implement a *stack* (of unlimited size), used in the calculator program in the exercise. Use the list implementation to implement the required stack operations and test your implementation with this program. Modify the struct declaration in `stack.h` and replace `stack.c` by your new implementation; do not modify `calc.c`.

2. **Flexible Arrays, using Braun Trees**

   Braun trees are binary trees where the subtrees have either the same number of nodes, or the left subtree has only one more, and each subtree is again a Braun tree. This is a simple, but very rigid condition: the shape of a tree is completely determined by the number of nodes in it.

Braun trees can be used to implement *flexible arrays*, arrays that can be enlarged on the left and on the right. A flexible arrays provides both a reasonably efficient lookup function to access elements at any position n between 0 and the size of the array, and efficient functions to add or remove elements on both left and right.

The picture on the side shows a Braun tree with 15 elements, and the numbering scheme for the lookup function. Note that all nodes with odd indices are in the left subtree.



For example, in order to add an element at the "front", it is inserted as the root. All indices are incremented by one, so the former right subtree becomes the new left subtree, with the former root data added at its top (recursively), and the former left subtree becomes the new right subtree. Complexity is logarithmic in the number of elements.

Shown below is a header file and parts of an implementation for flexible arrays that use Braun trees. The code is also provided on Absalon for download.

(a) Implement the functions `addL` (which is easy!) and `remvR` (which uses helper function `del`), following the hints in the source file. Take care to free all allocated memory when deleting and to properly allocate space for new elements in the heap.

(b) Write a test program for your implementation that inserts a configurable number of elements into a tree and and then deletes them, using all insert and remove functions.

braunseq.h

```
typedef double Data; /* defines the contained data type. Can be anything. */
typedef struct bNode_ {
  Data el;
  struct bNode_* left;
  struct bNode_* right;
} bNode;
typedef bNode** Tree; /* tree = a place to store a ptr to the root bNode */

int size(Tree tree);
void addL(Tree tree, Data new_el); /* append to left  */
void addR(Tree tree, Data new_el); /* append to right */
Data remvL(Tree tree); /* remove from left, return removed element  */
Data remvR(Tree tree); /* remove from right, return removed element */
Data lookup(Tree tree, int i); /* return element at index i (or NULL) */
```

Implementation braunseq.c

```
#include <stdlib.h> // standard library, defining NULL, malloc, free
#include <stdio.h>  // standard I/O functions

// data types and interface
#include "braunseq.h"

/*
 * internal helper functions
 *
 ***************************/
void error(char* msg) { // report an error and exit
  printf("PROGRAM ERROR: %s\n", msg);
  exit(EXIT_FAILURE);
}

// divide by 2 using bit shift
inline int half(int i) { return (i>>1); }
// test if an int is odd (return 1 if it is, 0 otherwise)
inline int odd(int i) { return (i & 0x1); }

// recursive helper for size: difference btw. tree size and m (recursive)
int diff(int n, bNode* t);
// insert at last position, knowing the tree size i >= 0 (for recursion)
void ins(Tree tree, int i, Data el);
```

```
// combine subtrees t1 and t2 into t1. Assumes size t1 same or 1 bigger than t2
void combine(Tree t1, Tree t2);
// delete last element, knowing the tree size i >= 0 (for recursion)
void del(Tree tree, int i);

int size(Tree tree) {
  if (*tree == NULL) {
    return 0;
  } else {
    int sz = size(&((*tree)->right));
    return (1 + 2*sz + diff(sz, (*tree)->left));
  }
}

void addL(Tree tree, Data new_el){
  // insert into new root, push old root into right subtree, swap sides
  return;
}

void addR(Tree tree, Data new_el) {
  // navigate to last element (use size), insert there
  ins(tree, size(tree), new_el);
}

Data remvL(Tree tree) {
  // remove root (use "combine" helper), return removed element
  bNode old_root = **tree; // make a copy
  free(*tree); // return unused memory
  *tree = old_root.left;
  combine(tree, &old_root.right);
  return old_root.el;
}

Data remvR(Tree tree) {
  // use size to navigate through the tree recursively (helper del)
  // before deleting, call lookup to retrieve the data that is returned
  return 0;
}

Data lookup(Tree tree, int i) {
  // navigate through tree recursively (i odd/even)
  if (*tree == NULL || i < 0) {
    error("lookup: index out of bounds");
  }
  if (i == 0) {
    return (*tree)->el;
  }
  if (odd(i)) {
    return ( lookup(&((*tree)->left), half(i)) );
  } else {
    return ( lookup(&((*tree)->right), half(i)-1) );
  }
}

/* ************************
 * Implementation internals
 *
 *************************/

// helper for size: difference btw. tree size and m (recursive, O(log n))
int diff(int n, bNode* t) {
  if (n == 0) { return (t == NULL ? 0 : 1); }
  // n != 0
  if (t == NULL) { error("diff encountered an internal error."); }
  if (odd(n)) {
    return diff(half(n), t->left);
  } else {
    return diff(half(n)-1, t->right);
  }
}

// insert at last position, knowing size is i >= 0 (recursive)
void ins(Tree tree, int i, Data el) {
  if (i < 0) { error("ins received invalid data."); }
```

```
  if (i == 0) {
    bNode* node;
    if (*tree != NULL) { error("ins expected tree==NULL"); }
    node = malloc(sizeof(bNode));
    if (node == NULL) { error("insert: failed to malloc"); }
    node->el = el;
    node->left = NULL;
    node->right=NULL;
    *tree = node;
    return;
  }
  if (odd(i)) { // if i odd: insert into left subtree
    ins(&((*tree)->left), half(i), el);
  } else {      // if even: insert into right subtree
    ins(&((*tree)->right), half(i)-1, el);
  }
}

// combine two subtrees into one. assumes size t1 same or 1 bigger than t2
void combine(Tree t1, Tree t2) {
  if (*t1 == NULL) { return; // t1 empty, so is t2. Nothing to do.
  } else {
    // otherwise, the new root is the t1 root, with left subtrees t2
    // and right subtree  combined from t1 subtrees (recursively)
    combine(&((*t1)->left), &((*t1)->right));
    (*t1)->right = (*t1)->left;
    (*t1)->left = *t2;
  }
}

// delete last element, knowing the size is i >= 0 (recursive)
void del(Tree tree, int i) {
  // navigate through the tree to find the last element (i is size).
  return;
}
```