Take Home Final - Computer Programming (0455-3391-01)

http://www.cs.tau.ac.il/~efif/courses/ComputerProgramming

Due: Febuary 9, 2003

We define a *set* as a group of elements on which the following operations are legal: (1) inserting an element into the group (2) removing an element from the group, and (3) inquiring whether an element is a member in the group. There are no multiple occurrences of elements in a set. Attempting to insert an element already in the set, or to remove an element not in the set, has no effect (no warning messages are necessary in these 2 cases).

1. Write a program that implements a set of non-negative integers. It reads commands provided by the user, and performs the corresponding operations immediately. The user commands are:

   help - print a message listing the commands. (could be this list).

   exit - clean up and exit the program

   print - print all elements of *set* in sorted order

   size - print the number of elements in *set*

   insert *e* - insert element *e* into *set*

   remove *e* - remove element *e* from *set*

   member *e* - is element *e* in *set*?

   Sample session:

   ```
   insert 8
   insert 1
   insert 7
   insert 2
   remove 7
   insert 6
   insert 3
   remove 3
   insert 5
   remove 1
   insert 4
   print
   2 4 5 6 8
   member 2
   2 is a member
   member 3
   3 is not a member
   exit
   ```

Define your own **struct** *Element*, and maintain a linked list of elements sorted in ascending order. Implement and use the following functions:

```
struct Element * insertElement(struct Element * firstElement, int elementId);
struct Element * removeElement(struct Element * firstElement, int elementId);
int memberElement(struct Element * firstElement, int elementId);
void printSet(struct Element * firstElement);
int sizeSet(struct Element * firstElement);
void cleanSet(struct Element * firstElement);
void help(void)
```

The first parameter to most of the functions above, namely *set*, points to the first element in the given set.

The functions *insertElement* and *removeElement* return the pointer to the first element of the resulting set. In addition, *insertElement* returns NULL, if something went wrong.

The function *sizeSet* returns the number of elements in the given set.

The function *cleanSet* cleans up, and should be used before exiting the program.

2. Extend your program from the previous exercise to handle the merging of two sets. In the previous exercise you assumed there was only one set and user commands applied to it. Now, assume there could be multiple sets, but only one of them is active. All previous opertions apply to the active set. Sets are identified by a non-negative integer. (If you wish, think of it as a set of sets).

   When the program starts it should create set 0 and make it active. The additional user commands sre:

   printAll - print all sets in ascending order

   set s - create set s if s does not exist already, and make set s active

   merge s - merge the elements of set s into the active set. Set s is removed as a result of this command

   Sample session:

```
set 0
insert 0
set 1
insert 1
set 2
insert 2
printAll
0: 0
1: 1
2: 2
set 0
merge 1
merge 2
printAll
0: 0 1 2
set 1
```

```
insert 1
insert 2
insert 3
set 0
merge 1
printAll
0: 0 1 2 3
exit
```

Update the **help** command to list the additional commands and their descriptions.

Define your own **struct** *Set*, and maintain a linked list of sets sorted in ascending order. Implement and use the following functions:

```
struct Set * makeSet(struct Set * firstSet, int setId, struct Set ** newSetPtr);
struct Set * mergeSet(struct Set * firstSet, struct Set * activeSet, int setId)
void printAllSets(struct Set * firstSet)
void cleanAllSets(struct Set * firstSet)
```

The function *makeSet* creates a new set with id *setId*, if a set with this id does not exist already. It returns the pointer to the set with the given id via the variable *newSetPtr*. This function takes as a parameter a pointer to the list of all sets *firstSet*, and returns a pointer to the list of all sets or NULL if something went wrong.

The function *mergeSet* merges the set identified by *setId* with the given *activeSet* into *activeSet*. The merged set identified by *setId* is removed as a result of this call. It returns a pointer to the list of all sets.

The function *cleanAllSets* cleans up, and should be used before exiting the program. The functions *printAllSets* and *cleanAllSets* should call *printSet* and *cleanSet* respectively in a loop.

3. Write a program to measure the performance of the merge operation. A program that gives a measure of performance is commonly referred as a *benchmark*.

   Implement the function

   ```
   void fillSet(struct Set * set, unsigned int size,
                unsigned int min, unsigned int max);
   ```

   This function fills the given set *set* with *size* integers randomly chosen from the range *min* to *max*. Use the *rand* function to draw the integers after setting the random-generator seed to *size* with the *srand* function.

   Use the function *fillSet* to generate a large number of sets each filled with random integers. Then, merge all the sets into set 0, and use the system funuction *clock* to time the sequence of merge operations. Finally, print out the results in seconds.

   Notice that the merge operation implemented naively by inserting the elements of the merged set into the active set one at a time, and cleaning the merged set at end, results with poor performance. Bonuses will be given according to code efficiency!

```
#define LOOP_SIZE 1000
  int i;
  clock_t time_start, time_end;
  float time_elapsed;
  struct Set *firstSet = NULL, *activeSet = NULL;

  for (i = LOOP_SIZE-1; i >= 0; --i) {
    firstSet = makeSet(firstSet, i, &activeSet);
    fillSet(activeSet, i, 0, i-1);
  }
  time_start = clock();
  for (i = 1; i < LOOP_SIZE; ++i) {
    firstSet = mergeSet(firstSet, activeSet, i);
  }
  time_end = clock();
  time_elapsed = (float) (time_end - time_start) / (float) CLOCKS_PER_SEC;
  printf("Average operation time in sec. = %f, \n", time_elapsed / LOOP_SIZE);
```

NAME	**rand**, **srand** - random number generator.

SYNOPSIS	`#include <stdlib.h>`
	`int rand(void);`
	`void srand(unsigned int seed);`

DESCRIPTION	The **rand()** function returns a pseudo-random integer between 0 and **RAND_MAX**. The **srand()** function sets its argument as the seed for a new sequence of pseudo-random integers to be returned by *rand()*. These sequences are repeatable by calling **srand()** with the same seed value. If no seed value is provided, the **rand()** function is automatically seeded with a value of 1.
RETURN VALUE	The **rand()** function returns a value between 0 and **RAND_MAX**. The **srand()** function returns no value.

NAME	**clock** - Determine processor time
SYNOPSIS	`#include <time.h>` `clock_t clock(void);`
DESCRIPTION	The **clock()** function returns an approximation of processor time used by the program.
RETURN VALUE	The value returned is the CPU time used so far as a **clock_t**; to get the number of seconds used, divide by **CLOCKS_PER_SEC**.

4. Implement an operation that prints all sets in descending order, and all elements of each set in descending order. The user command is:

printRevAll - print all elements is reverse order