CSE310 Project 1: Maxheap, Modular Design, and File IO
Posted: Wednesday, 02/19/2020, Due: Monday, 03/16/2020

This is a programming project in either C++ or C, to be completed and graded on general.asu.edu, a Linux machine. You will perform modular design, provide a Makefile to compile various modules to generate the executable file named run. Among other things, you need to have

1. a main program, which coordinates all other modules;
2. a module that provides utility services including command line interpretation;
3. a module that implements the max-heap data structure;
4. a Makefile which compiles all modules and link them into the executable.

For each module other than the main program, you should have a header file which specifies the prototypes of the functions in the module, and an implementation file which implements all of the functions specified in the header file. Your program should be based on the g++ compiler (for C++) or the gcc compiler (for C) on general.asu.edu. **All programs will be compiled and graded on general.asu.edu**. You will need to submit it electronically on Canvas, in one zip file, named CSE310-P01-Lname-Fname, where Lname is your last name and Fname is your first name. The zip file should contain a minimum set of files that are absolutely necessary to compile and execute your program (it should include the Makefile, the header files and the implementation files, but not the object (.o) files). **If your program does not compile and work on general.asu.edu, you will receive 0 on this project**.

You need to define the following data types.

- ELEMENT is a data type that contains a field named key, which is of type int. Note that ELEMENT should not be of type int.
- HEAP is a data type that contains three fields named capacity (of type int), size (of type int), and H (an array of type ELEMENT with index ranging from 0 to capacity).

The functions that you are required to implement are:

- Initialize(n) which returns an object of type HEAP with capacity n and size 0. This function requires you to perform dynamic memory allocation, given the demand n.
- BuildHeap(heap, A, n), where heap is a HEAP object, A is an array of type ELEMENT, and n is the size of array A. This function copies the elements in A into `heap->H` (starting from `H[1]` and uses the linear time build heap algorithm to obtain a max-heap of size n from the given array A.

- Insert(heap, flag, k) which inserts an element with key equal to k into the max-heap heap. When flag=1, the function does not do any additional printing. When flag=2, the function prints out the heap content before the insertion, and the heap content after the insertion.
- DeleteMax(heap, flag) which deletes the element with maximum key and returns it to the caller. When flag=1, the function does not do any additional printing. When flag=2, the function prints out the heap content before the deletion, and the heap content after the deletion.
- IncreaseKey(heap, flag, index, value) which increases the key field of the heap element pointed to by index to value, which should not be smaller than the current value. Note that you have to make necessary adjustment to make sure that heap order is maintained. When flag=1, the function does not do any additional printing. When flag=2, the function prints out the heap content before the increase key operation, and the heap content after the increase key operation.
- printHeap(heap) which prints out the heap information, including capacity, size, and the key fields of the elements in the array with index going from 1 to size.

You should implement a module that takes the following commands from the key-board and feeds to the main program:

- **S**
- **C n**
- **R**
- **W**
- **I f k**
- **D f**
- **K f i v**

On reading **S**, the program stops.
On reading **C n**, the program creates an empty heap with capacity equal to **n**, and waits for the next command.
On reading **R**, the program reads in the array A from file HEAPinput.txt, calls the linear time build heap algorithm to build the max-heap based on A, and waits for the next command.
On reading **W**, the program writes the current heap information to the screen, and waits for the

next command. The output should be in the same format as in the file `HEAPinput.txt`, proceeded by the heap capacity.
On reading **I f k**, the program inserts an element with key equal to **k** into the current heap with the corresponding flag set to **f**, and waits for the next command.
On reading **D f**, the program deletes the maximum element from the heap with the corresponding flag set to **f**, and prints the key field of the deleted element on the screen, it waits for the next command.
On reading **K f i v**, the program increases the key of element with index i to v with the corresponding flag set to **f**.

The file HEAPinput.txt is a text file. The first line of the file contains an integer n, which indicates the number of array elements. The next n lines contain n integers, one integer per line. These integers are the key values of the n array elements, from the first element to the nth element.

Grading policies: (Sample test cases will be posted soon.) All programs will be compiled (using the Makefile you provided) and executed on general.asu.edu. If your program does not compile and execute on general.asu.edu, you will receive 0 for this project. So start working today, and do not claim "my program works perfectly on my PC, but I do not know how to use general.asu.edu."

(10 pts) You should provide a Makefile that can be used to compile your project on general.asu.edu. The executable file should be named run. If your program does not pass this step, you will receive 0 on this project.

(10 pts) Modular design: You should have a file named util.cpp and its corresponding header file util.h, where the header file defines the prototype of the functions, and the implementation file implements the functions. You should have a file named heap.cpp and its corresponding header file heap.h. This module implements the heap functions.

(10 pts) Documentation: You should provide sufficient comment about the variables and algorithms. You also need to provide a README file describing which language you are using.

(10 pts) Your program should use dynamic memory allocation correctly.

(30 pts) Your program should produce the correct output for the posted set of test cases.

(30 pts) Your program should produce the correct output for an unposted set of test cases.

You should try to make your program as robust as possible. A basic principle is that your program can complain about bad input, but should not crash. When you need to increase the

capacity of the heap, try to increase it to the smallest power of 2 that is large enough for your need. If you can use the realloc command to avoid copying the array. If that is not successful, then allocate a new piece of memory.

As an aid, the following is a partial program for reading in the commands from the keyboard. You need to understand it and to expand it.

```
#include "util.h"
//=========================================================================
int nextCommand(int *i, int *v, int *f)
{
  char c;
  while(1){
    scanf("%c", &c);
    if (c == ' ' || c == '\t' || c == '\n'){
        continue;
    }
    if (c == 'S' || c == 'R' || c == 'W'){
        break;
    }
    if (c == 'K' || c == 'k'){
        scanf("%d", i); scanf("%d", v); scanf("%d", f);
        break;
    }
    if (...){
    ...
    }
    printf("Invalid Command\n");
  }
  return c;
}
//=========================================================================
```

The following is a partial program that calls the above program.

```
//========================================================================
#include <stdio.h>
#include <stdlib.h>
#include "util.h"

int main()
{
    // variables for the parser...
    char c;
    int i, v;
    while(1){
        c = nextCommand(&i, &v, &f);
        switch (c) {
            case 's':
            case 'S': printf("COMMAND: %c.\n", c); exit(0);

            case 'k':
            case 'K': printf("COMMAND: %c %d %d %d.\n", c, i, v, f); break;

            default: break;
        }
    }
    exit(0);
}
//========================================================================
```

The following is a partial Makefile.

```
EXEC = run
CC = g++
CFLAGS = -c -Wall

# $(EXEC) has the value of shell variable EXEC, which is run.
# run depends on the files main.o util.o heap.o
$(EXEC) :main.o util.o heap.o
```

```
# run is created by the command g++ -o run main.o util.o
# note that the TAB before $(CC) is REQUIRED...
        $(CC) -o $(EXEC) main.o util.o heap.o

# main.o depends on the files main.h main.cpp
main.o:main.h main.cpp
# main.o is created by the command g++ -c -Wall main.cpp
# note that the TAB before $(CC) is REQUIRED...
        $(CC) $(CFLAGS) main.cpp

util.o  :util.h util.cpp
        $(CC) $(CFLAGS) util.cpp

heap.o  :heap.h heap.cpp
        $(CC) $(CFLAGS) heap.cpp

clean   :
        rm *.o
```