มหาวิทยาลัยสงขลานครินทร์

คณะวิศวกรรมศาสตร์ ภาควิชาวิศวกรรมคอมพิวเตอร์

ข้อสอบปลายภาคการศึกษาที่ 1 ประจำปีการศึกษา 2550

วันอาทิตย์ที่ 7 ตุลาคม พ.ศ. 2550 เวลา 13.30 - 16.30 น.

วิชา 240-204 และ 241-207 Data Structure and Computer Programming Techniques ห้องสอบ R200, R201

คำสั่ง

- ข้อสอบมีทั้งหมด 10 ข้อ 10 หน้า (รวมหน้านี้) รวมคะแนน 47 คะแนน ให้ทำทุกข้อ
- ไม่อนุญาตให้นำเครื่องคิดเลขและเอกสารใดๆ เข้าห้องสอบ
- ให้เขียนชื่อ-นามสกุล และรหัสนักศึกษาไว้ทุกหน้า
- ให้ตอบคำถามลงในข้อสอบ หากมีที่ว่างไม่พอให้ใช้พื้นที่ด้านหลังของข้อนั้นๆเท่านั้น
- ให้กระดาษทด 1 แผ่น อยู่ด้านหลังข้อสอบ อนุญาตให้ดึงออกได้ แต่ควรระวังไม่ให้ข้อสอบติดออกมาด้วย
- อนุญาตให้ใช้ดินสอตอบคำถามได้

ชื่อ-นามสกุล..รหัสนักศึกษา............................

ข้อ	คะแนน
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	
รวม	

ชื่อ-นามสกุล..รหัสนักศึกษา..

คำอธิบายการใช้งานฟังก์ชันที่สำคัญ

Function: int **fclose** (*FILE *stream*) is declared in **stdio.h**.

This function causes *stream* to be closed and the connection to the corresponding file to be broken. Any buffered output is written and any buffered input is discarded. The `fclose` function returns a value of `0` if the file was closed successfully, and `EOF` if an error was detected.

It is important to check for errors when you call `fclose` to close an output stream, because real, everyday errors can be detected at this time. For example, when `fclose` writes the remaining buffered output, it might get an error because the disk is full. Even if you know the buffer is empty, errors can still occur when closing a file if you are using NFS.

To close all streams currently available the GNU C Library provides another function.

Function: FILE * **fopen** (*const char *filename, const char *opentype*) is declared in **stdio.h**.

The `fopen` function opens a stream for I/O to the file *filename*, and returns a pointer to the stream.

The *opentype* argument is a string that controls how the file is opened and specifies attributes of the resulting stream. It must begin with one of the following sequences of characters:

`r' Open an existing file for reading only.

`w' Open the file for writing only. If the file already exists, it is truncated to zero length. Otherwise a new file is created.

`a' Open a file for append access; that is, writing at the end of file only. If the file already exists, its initial contents are unchanged and output to the stream is appended to the end of the file. Otherwise, a new, empty file is created.

`r+' Open an existing file for both reading and writing. The initial contents of the file are unchanged and the initial file position is at the beginning of the file.

`w+' Open a file for both reading and writing. If the file already exists, it is truncated to zero length. Otherwise, a new file is created.

`a+' Open or create file for both reading and appending. If the file exists, its initial contents are unchanged. Otherwise, a new file is created. The initial file position for reading is at the beginning of the file, but output is always appended to the end of the file.

As you can see, `+' requests a stream that can do both input and output. The ISO standard says that when using such a stream, you must call `fflush` (see Stream Buffering) or a file positioning function such as `fseek` (see File Positioning) when switching from reading to writing or vice versa. Otherwise, internal buffers might not be emptied properly. The GNU C library does not have this limitation; you can do arbitrary reading and writing operations on a stream in whatever order.

Additional characters may appear after these to `specify` flags for the call. Always put the mode (`r', `w+', etc.) first; that is the only part you are guaranteed will be understood by all systems.

The GNU C library defines one additional character for use in *opentype*: the character `x' insists on creating a new file—if a file *filename* already `exists`, `fopen` fails rather than opening it. If you use `x' you are guaranteed that you will not clobber an existing file. This is equivalent to the `O_EXCL` option to the `open` function (see Opening and Closing Files).

The character `b' in *opentype* has a standard meaning; it requests a binary stream rather than a text stream. But this makes no difference in POSIX systems (including the GNU system). If both `+' and `b' are specified, they can appear in either order. See Binary Streams.

If the *opentype* string contains the sequence `,ccs=`*STRING* then *STRING* is taken as the name of a coded character set and `fopen` will mark the stream as wide-oriented which appropriate conversion functions in place to convert from and to the character set *STRING* is place. Any other stream is opened initially unoriented and the orientation is decided with the first file operation. If the first operation is a wide character operation, the stream is not only marked as wide-oriented, also the conversion functions to convert to the coded character set used for the current locale are loaded. This will not change anymore from this point on even if the locale selected for the `LC_CTYPE` category is changed.

Any other characters in *opentype* are simply ignored. They may be meaningful in other systems.

If the open fails, `fopen` returns a null pointer.

When the sources are compiling with `_FILE_OFFSET_BITS == 64` on a 32 bit machine this function is in fact `fopen64` since the LFS interface replaces transparently the old interface.

You can have multiple streams (or file `descriptors`) pointing to the same file open at the same time. If you do only

input, this works straightforwardly, but you must be careful if any output streams are included. See Stream/Descriptor Precautions. This is equally true whether the streams are in one program (not usual) or in several programs (which can easily happen). It may be advantageous to use the file locking facilities to avoid simultaneous access. See File Locks.

Function: int **fprintf** (*FILE *stream, const char *template, ...*) is declared in **stdio.h**.
This function is just like `printf`, except that the output is written to the stream *stream* instead of `stdout`.

Function: int **fputc** (*int c, FILE *stream*) is declared in **stdio.h**.
The `fputc` function converts the character *c* to type `unsigned char`, and writes it to the stream *stream*. `EOF` is returned if a write error occurs; otherwise the character *c* is returned.

Function: int **fputs** (*const char *s, FILE *stream*) is declared in **stdio.h**.
The function `fputs` writes the string *s* to the stream *stream*. The terminating null character is not written. This function does *not* add a newline character, either. It outputs only the characters in the string.

This function returns `EOF` if a write error occurs, and otherwise a non-negative value. For example:

```
fputs ("Are ", stdout);
fputs ("you ", stdout);
fputs ("hungry?\n", stdout);
```

outputs the text `` `Are you hungry?' `` followed by a newline.

Function: size_t **fread** (*void *data, size_t size, size_t count, FILE *stream*) is declared in **stdio.h**.
This function reads up to *count* objects of size *size* into the array *data*, from the stream *stream*. It returns the number of objects actually read, which might be less than *count* if a read error occurs or the end of the file is reached. This function returns a value of zero (and doesn't read anything) if either *size* or *count* is zero.

If `fread` encounters end of file in the middle of an object, it returns the number of complete objects read, and discards the partial object. Therefore, the stream remains at the actual end of the file.

Function: int **fscanf** (*FILE *stream, const char *template, ...*) is declared in **stdio.h**.
This function is just like `scanf`, except that the input is read from the stream *stream* instead of `stdin`.

Function: int **fseek** (*FILE *stream, long int offset, int whence*) is declared in **stdio.h**.
The `fseek` function is used to change the file position of the stream *stream*. The value of *whence* must be one of the constants `SEEK_SET`, `SEEK_CUR`, or `SEEK_END`, to indicate whether the *offset* is relative to the beginning of the file, the current file position, or the end of the file, respectively.

This function returns a value of zero if the operation was successful, and a nonzero value to indicate failure. A successful call also clears the end-of-file indicator of *stream* and discards any characters that were "pushed back" by the use of `ungetc`.

`fseek` either flushes any buffered output before setting the file position or else remembers it so it will be written later in its proper place in the file.

Function: size_t **fwrite** (*const void *data, size_t size, size_t count, FILE *stream*) is declared in **stdio.h**.
This function writes up to *count* objects of size *size* from the array *data*, to the stream *stream*. The return value is normally *count*, if the call succeeds. Any other value indicates some sort of error, such as running out of space.

Function: int **printf** (*const char *template, ...*) is declared in **stdio.h**.
The `printf` function prints the optional arguments under the control of the template string *template* to the stream `stdout`. It returns the number of characters printed, or a negative value if there was an output error.

Function: int **putc** (*int c, FILE *stream*) is declared in **stdio.h**.
This is just like `fputc`, except that most systems implement it as a macro, making it faster. One consequence is that it may evaluate the *stream* argument more than once, which is an exception to the general rule for macros. `putc` is usually the best function to use for writing a single character.

Function: int **puts** (*const char *s*) is declared in **stdio.h**.
The `puts` function writes the string *s* to the stream `stdout` followed by a newline. The terminating null character of the string is not written. (Note that `fputs` does *not* write a newline as this function does.)

`puts` is the most convenient function for printing simple messages. For example:

```
puts ("This is a message.");
```

outputs the text `` `This is a message.' `` followed by a newline.

Function: int **scanf** (*const char *template, ...*) is declared in **stdio.h**.
The `scanf` function reads formatted input from the stream `stdin` under the control of the template string

template. The optional arguments are pointers to the places which receive the resulting values.

The return value is normally the number of successful assignments. If an end-of-file condition is detected before any matches are performed, including matches against whitespace and literal characters in the template, then `EOF` is returned.

Function: int **sprintf** (*char *s, const char *template, ...*) is declared in **stdio.h**.

This is like `printf`, except that the output is stored in the character array *s* instead of written to a stream. A null character is written to mark the end of the string.

The `sprintf` function returns the number of characters stored in the array *s*, not including the terminating null character.

The behavior of this function is undefined if copying takes place between objects that overlap—for example, if *s* is also given as an argument to be printed under control of the `` `%s' `` conversion.

Warning: The `sprintf` function can be **dangerous** because it can potentially output more characters than can fit in the allocation size of the string *s*. Remember that the field width given in a conversion specification is only a *minimum* value.

To avoid this problem, you can use `snprintf` or `asprintf`, described below.

Function: int **sscanf** (*const char *s, const char *template, ...*) is declared in **stdio.h**.

This is like `scanf`, except that the characters are taken from the null-terminated string *s* instead of from a stream. Reaching the end of the string is treated as an end-of-file condition.

The behavior of this function is undefined if copying takes place between objects that overlap—for example, if *s* is also given as an argument to receive a string read under control of the `` `%s' ``, `` `%S' ``, or `` `%[' `` conversion.

Function: int **strcmp** (*const char *s1, const char *s2*) is declared in **string.h**.

The `strcmp` function compares the string *s1* against *s2*, returning a value that has the same sign as the difference between the first differing pair of characters (interpreted as `unsigned char` objects, then promoted to `int`).

If the two strings are equal, `strcmp` returns `0`.

A consequence of the ordering used by `strcmp` is that if *s1* is an initial substring of *s2*, then *s1* is considered to be "less than" *s2*.

`strcmp` does not take sorting conventions of the language the strings are written in into account. To get that one has to use `strcoll`.

Function: char * **strcpy** (*char *restrict to, const char *restrict from*) is declared in **string.h**.

This copies characters from the string *from* (up to and including the terminating null character) into the string *to*. Like `memcpy`, this function has undefined results if the strings overlap. The return value is the value of *to*.

Function: size_t **strlen** (*const char *s*) is declared in **string.h**.

The `strlen` function returns the length of the null-terminated string *s* in bytes. (In other words, it returns the offset of the terminating null character within the array.) For example,

```
strlen ("hello, world")

    => 12
```

When applied to a character array, the `strlen` function returns the length of the string stored there, not its allocated size. You can get the allocated size of the character array that holds a string using the `sizeof` operator:

```
char string[32] = "hello, world";
sizeof (string)
    => 32
strlen (string)
    => 12
```

But beware, this will not work unless *string* is the character array itself, not a pointer to it. For example:

```
char string[32] = "hello, world";
char *ptr = string;
sizeof (string)
    => 32
sizeof (ptr)
    => 4  /* (on a machine with 4 byte pointers) */
```

ชื่อ-นามสกุล..รหัสนักศึกษา..

This is an easy mistake to make when you are working with functions that take string arguments; those arguments are always pointers, not arrays.

It must also be noted that for multibyte encoded strings the return value does not have to correspond to the number of characters in the string. To get this value the string can be converted to wide characters and `wcslen` can be used or something like the following code can be used:

```
/* The input is in string. The length is expected in n.  */
{
  mbstate_t t;
  char *scopy = string;
  /* In initial state.  */
  memset (&t, '\0', sizeof (t));
  /* Determine number of characters.  */
  n = mbsrtowcs (NULL, &scopy, strlen (scopy), &t);
}
```

This is cumbersome to do so if the number of characters (as opposed to bytes) is needed often it is better to work with wide characters.

ข้อสอบ

1. ในการเก็บข้อมูลของนักศึกษาที่เรียนวิชาหนึ่ง มีข้อมูลที่จำเป็นต้องเก็บไว้คือ **รหัสนักศึกษา (ประกอบไปด้วยตัวเลขหรือตัวอักษร 10 หลัก), คะแนนรวม และเกรด** จงเขียนโครงสร้างข้อมูลนี้ โดยกำหนดให้ใช้ชื่อ **Info** พร้อมทั้งอธิบายเหตุผลประกอบ **(1 คะแนน, 2 นาที)**

struct Info { **}**	**คำอธิบาย:**

2. จากโครงสร้างข้อมูล **Info** หากต้องการเก็บข้อมูลนักศึกษาหลายๆคนในแบบ List และ Tree **โครงสร้างข้อมูลของแต่ละโหนดควรจะเป็นอย่างไร** ให้โครงสร้างข้อมูลแบบ List ชื่อ **InfoList** และแบบ Tree ชื่อ **InfoTree (2 คะแนน, 4 นาที)**

struct InfoList { **}**	**struct InfoTree {** **}**

ทุจริตในการสอบ โทษขั้นต่ำคือ ปรับตกในรายวิชาที่ทุจริต และพักการเรียน 1 ภาคการศึกษา

ชื่อ-นามสกุล...รหัสนักศึกษา..

3. จากโครงสร้างข้อมูล **InfoList** จงเขียนฟังก์ชันสำหรับเพิ่มข้อมูลต่อท้าย List ที่กำหนดให้ (ใช้การจองหน่วยความจำใหม่) และ **ถ้า List ที่กำหนดให้มีค่าเป็น NULL เมื่อฟังก์ชันจบการทำงานแล้ว ตำแหน่งของ List ที่กำหนดให้ จะต้องถูกเปลี่ยนให้ชี้ไปที่ข้อมูลใหม่ที่สร้างขึ้น** โดยให้ตั้งชื่อฟังก์ชันเป็น ListAdd และเวลาเรียกใช้จะเรียกใช้ดังนี้ **(4 คะแนน, 12 นาที)**

```
struct InfoList *head = NULL;
ListAdd(&head, รหัสนักศึกษา, คะแนนรวม, เกรด);
```

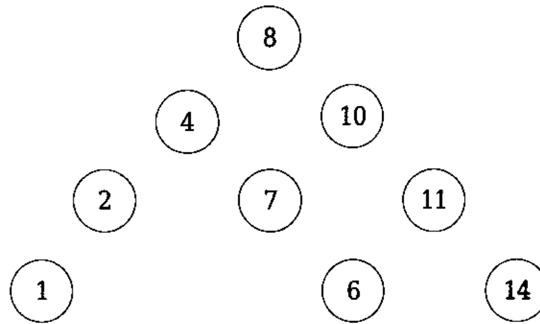
4. จงเขียนฟังก์ชันที่ทำการค้นหาข้อมูลใน List ที่แต่ละโหนดมีโครงสร้างเป็น **InfoList** โดย**ใช้รหัสนักศึกษาเป็นค่าหลัก**ในการค้นหา และให้ค้นหาแบบ Linear Search โดยที่**ฟังก์ชันนี้จะ return ค่าตำแหน่งของข้อมูลที่ค้นพบ** ถ้าหากค้นหาไม่พบ ให้ return ค่าเป็น NULL **(3 คะแนน, 6 นาที)**

5. จากโครงสร้างข้อมูล **InfoTree** จงเขียนฟังก์ชันสำหรับเพิ่มข้อมูลเข้าไปใน Tree ที่กำหนดให้ในแบบ Binary Search Tree (ใช้การจองหน่วยความจำใหม่) **โดยให้ใช้รหัสนักศึกษาเป็นหลักในการพิจารณาสร้าง Tree และถ้า Tree ที่กำหนดให้มีค่าเป็น NULL เมื่อฟังก์ชันจบการทำงานแล้ว ตำแหน่งของ Tree ที่กำหนดให้ จะต้องถูกเปลี่ยนให้ชี้ไปที่ข้อมูลใหม่ที่สร้างขึ้น** โดยให้ตั้งชื่อฟังก์ชันเป็น IDBSTAdd และเวลาเรียกใช้ จะเรียกใช้ดังนี้ **(5 คะแนน, 15 นาที)**

```
struct InfoTree *root = NULL;
IDBSTAdd(&root, รหัสนักศึกษา, คะแนนรวม, เกรด);
```

ชื่อ-นามสกุล...รหัสนักศึกษา..

6. จงลากเส้นเชื่อมโยงระหว่างโหนดแต่ละโหนดให้เป็นแผนภาพต้นไม้แบบ BST **(1 คะแนน, 1 นาที)**



และแสดงผลลัพธ์เมื่อเรียกใช้ฟังก์ชันที่เดินไปใน BST เพื่อแสดงผลค่าแต่ละโหนดด้วยวิธีการแต่ละแบบ **(3 คะแนน, 6 นาที)**

Pre-order:____

In-order:____

Post-order:____

7. จงดัดแปลงฟังก์ชัน preorder_print ต่อไปนี้ ให้เป็น In-order traversal สำหรับโครงสร้างข้อมูล **InfoTree** ในข้อ 2

```
void preorder_print(TREE t) {
  if (!isEmptyTree(t)) {
    printf("%3d", value(t));    preorder_print(lsTree(t));    preorder_print(rsTree(t));
  }
}
```

และแทนที่จะแสดงผลค่าของโหนด ให้เปลี่ยนเป็นเขียนค่าทั้งหมดลงไฟล์ ซึ่งข้อมูลแต่ละโหนดในไฟล์ จะต้องสามารถอ่านกลับมาได้ด้วยคำสั่ง fread() โดยให้ฟังก์ชันนี้ชื่อว่า InOrderWrite และมีรูปแบบการเรียกใช้ดังตัวอย่างข้างล่างนี้ **(5 คะแนน, 15 นาที)**

InOrderWrite(ตำแหน่งของ root โหนด, ชื่อไฟล์);

8. จากฟังก์ชันค้นหาข้อมูลในโครงสร้างข้อมูลแบบ Binary Search Tree ต่อไปนี้ ถ้าถือว่าการเรียกตัวเองแต่ละครั้งเปรียบเสมือนการวนซ้ำหนึ่งครั้ง และไม่นับการตรวจสอบโหนดว่าเป็น empty โหนดหรือไม่ จงวิเคราะห์ความซับซ้อนของวิธีการค้นหาแบบนี้

```
int BSTSearch(int key, BST t) {
   int v;
   if (isEmptyTree(t))
      return 0;
   v = value(t);
   if (key == v)
      return 1;
   else if (key < v)
      return bstsearch(key, lsTree(t));
   else if (key > v)
      return bstsearch(key, rsTree(t));
}
```

Best Case คือกรณีที่ **(1 คะแนน, 1 นาที)**

____

____

____

C_{min} = ______________________ **(1 คะแนน, 1 นาที)**

Worst Case คือกรณีที่ **(3 คะแนน, 6 นาที)**

____

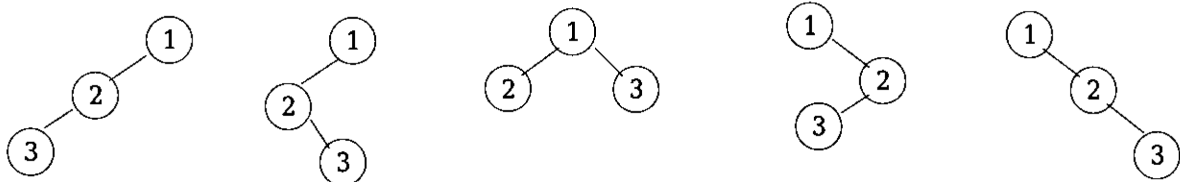
____

____

____

C_{max} = ______________________ **(2 คะแนน, 4 นาที)**

ถ้ามีข้อมูลอยู่ทั้งหมด 3 โหนด รูปแบบการเชื่อมโยง Tree ที่เป็นไปได้ทั้งหมดมีลักษณะดังรูปข้างล่างนี้ ในการค้นหาแล้วพบข้อมูลที่โหนด 1 หรือ 2 หรือ 3 ก็จะมีจำนวนครั้งในการเปรียบเทียบที่แตกต่างกันไป



จำนวนกรณีที่ค้นหาแล้วพบข้อมูลที่ โหนด 1 = ________, โหนด 2 = ________, โหนด 3 = ________, ไม่พบ = ________

รวมเป็นกรณีในการหาข้อมูลทั้งหมด = ______________________ **(2 คะแนน, 4 นาที)**

จำนวนครั้งในการเปรียบเทียบข้อมูลทั้งหมด = ______________________ **(2 คะแนน, 4 นาที)**

ค่าเฉลี่ยของจำนวนครั้งในการเปรียบเทียบ ($C_{average}$) = ______________________ **(1 คะแนน, 2 นาที)**

ชื่อ-นามสกุล...รหัสนักศึกษา...

9. ความซับซ้อนของการค้นหาข้อมูลใน Binary Search Tree ดังในข้อ 8 กับการค้นหาข้อมูลในอาเรย์แบบ Binary Search เหมือน หรือต่างกันหรือไม่ อย่างไร **(2 คะแนน, 6 นาที)**

10.จากลำดับข้อมูลเบื้องต้นในตาราง จงเขียนลำดับที่เปลี่ยนไปในแต่ละรอบของการจัดเรียงข้อมูลด้วยวิธีการ sort แบบต่างๆ เช่น Selection Sort, Insertion Sort, Bubble Sort แบบ native, Bubble Sort แบบ improve, และ Quick Sort โดยให้เลือกวิธีใดก็ได้ที่ท่านถนัดที่สุดมา 3 วิธี โดยให้ใส่ชื่อวิธีไว้ที่ช่องแถวบนสุดของตาราง **(วิธีละ 3 = 9 คะแนน, 18 นาที)**

ชื่อวิธี			
รอบที่	7, 5, 2, 9, 4, 1, 6, 0, 3, 8	7, 5, 2, 9, 4, 1, 6, 0, 3, 8	7, 5, 2, 9, 4, 1, 6, 0, 3, 8
1			
2			
3			
4			
5			
6			
7			
8			
9			
10			
11			
12			
13			
14			
15			
16			
17			