Test2aa

1. [25] Write a C function with prototype:
int insrec(char *fdb, int num, char *rec, int rsz);
This function will insert the record stored at memory location rec AFTER record number num in the file whose name is given by fdb. Notice that a hole in the file must be made in order to insert the new record in its current position. You may assume that the file contains at least num records. Furthermore you may assume that the file consist of an integral number of records and that all records have the same size, rsz. A record is merely a consecutive block of bytes of size rsz. The record indexing starts at 0 so record 0 consists of bytes 0 through rsz − 1, record 1 consists of bytes rsz through 2rsz − 1, and so on. The function must both open and close the file and read/writes must be record sized. If successful, insrec() will return a 0 else insrec() will return -1. The function must use low-level I/O except for printing error messages and you may assume that rsz ≤ 1024.

Data Members:

loop variable, file descriptor, a file size variable, and a buffer for records,

Outline:

1. open the file O_RDWR
2. get size of file (lseek to end)
3. create a hole buy moving recs down one for I = fsize/rsz -1 to > rnum, I--
    1. go up one record: lseek (I-1)*rsz and read
    2. file pointer should be at end of record, so just write
4. lseek to (rnum +1)*rsz and write the given record in the hole
5. close the file and return 0

You can assume every open, lseek, read, and write needs to be properly error trapped!

```
int insrec(char *fdb, int num, char *rec, int rsz) {
      int i;                                    // loop variable
      int fd;                                   // file descriptor
      int fsize;                                // file size
      char buf[rsz];                            // buffer to hold a record
      fd = open(fdb, O_RDWR);                   // open the file
      fsize = lseek(fd, 0, SEEK_END);           // get file size
      for(i = fsize/rsz - 1; i > num; i--){     // create hole - start at bottom
            lseek(fd, (i-1) * rsz, SEEK_SET);   // seek to record to be moved
            read(fd, buf, rsz);                 // read the record
            write(fd, buf, rsz);                // write the record - next lower
      }
      lseek(fd, (num+1)*rsz, SEEK_SET);         // seek to hole created
      write(fd, rec, rsz);                      // write the given record
      close(fd);                                // close file and return
      return 0;                                 // return 0 - all good
}
```

Test2bb

1. [25] Write a C function with prototype:
int delrecs(char *fdb, int rnum, int num, int rsz);
This function will delete the num records, starting with record rnum in the file whose name is given by fdb. Notice that there cannot be holes in the file, i.e. the resulting file must contain the remaining records as contiguous records. You may assume that the file contains at least rnum + num records. Furthermore you may assume that the file consist of an integral number of records and that all records have the same size, rsz. A record is merely a consecutive block of bytes of size rsz. The record indexing starts at 0 so record 0 consists of bytes 0 through rsz − 1, record 1 consists of bytes rsz through 2rsz − 1, and so on. The function must both open and close the file and read/writes must be record sized. If successful, delrecs() will return a 0 else delrecs() will return -1. The function must use low-level I/O except for printing error messages and you may assume that rsz ≤ 1024.

Data Members:
loop variable, file descriptor, buffer for record, file size variable

Outline:

1. open file O_RDWR
2. get file size (lseek to SEEK_END)
3. loop over records to keep (starting from rnum+num to fsize/rsz-1)
    1. seek to record to keep (i*rsz)
    2. read record
    3. seek to place to write (i-num)*rsz
    4. write record
4. truncate file (ftruncate(fd, fsize-num*rsz)
5. close file and return 0

You can assume every open, lseek, read, and write needs to be properly error trapped!

```
int delrecs(char *fdb, int rnum, int num, int rsz){
      int i;                                    // loop variables
      int fd;                                   // file descriptor
      char *buf[rsz];                           // buffer to keep record in
      int fsize;                                // size of file
      fd = open(fdb, O_RDWR);                   // open the file
      fsize = lseek(fd, 0, SEEK_END)            // get file size
      for(i = rnum+num; i < fsize/rsz -1; i++){ // loop over record after rnum + num
            lseek(fd, i*rsz, SEEK_SET);         // seek to record to keep
            read(fd, buf, rsz);                 // read the record
            lseek(fd, (i - num)*rsz, SEEK_SET); // seek up to the place to write
            write(fd, buf, rsz);                // write the record
      }
      ftruncate(fd, fsize - num*rsz);           // truncate the file
      close(fd);                                // close file
      return 0;                                 // return 0 - all good
}
```

Test2cc
1. [25] Write a C function with prototype:
int insrec(char *fdb1, int rnum, char *fdb2, int num, int rsz);
This function will read the record num from the file fdb2 and insert it into record position rnum in file fdb1. No record should be overwritten, the new record is simply inserted into that position leaving the order of all other records, before and after, the same. Furthermore you may assume that the files consist of an integral number of records and that all records have the same size, rsz. A record is merely a consecutive block of bytes of size rsz. The record indexing starts at 0 so record 0 consists of bytes 0 through rsz − 1, record 1 consists of bytes rsz through 2rsz − 1, and so on. The function must both open and close the file and read/writes must be record sized. If successful, insrec() will return a 0 else insrec() will return -1. The function must use low-level I/O except for printing error messages and you may assume that rsz ≤ 1024.

Data Members:
loop variable, two file descriptors, a buffer for a record, file fdb1 size variable

Outline:
1. open both files (fdb1 with O_RDWR and fdb2 with O_RDONLY)
2. get size of fdb1, (lseek SEEK_END)
3. make a hole in fdb1 (loop from fsize/rsz – 1 to rnum)
    1. lseek to (I-1)*rsz
    2. read record
    3. write record
4. read record from fdb2 (lseek, read)
5. write record in hole in fdb1 (lseek, write)
6. close both files and return 0

You can assume every open, lseek, read, and write needs to be properly error trapped!

```
int insrec(char *fdb1, int rnum, char *fdb2, int num, int rsz) {
      int i;                                       // loop variable
      int fd1;                                     // file descriptor for fdb1
      int fd2;                                     // file descriptor for fdb2
      int fsize;                                   // size of fdb1
      char buf[rsz];                               // buffer for record
      fd1 = open(fdb1, O_RDWR);                    // open the files
      fd2 = open(fdb2, O_RDONLY);
      fsize = lseek(fd1, 0, SEEK_END);             // get fdb1 size
      for(i = fsize/rsz - 1; i > rnum; i--){       // create hole in fdb1 – start at bottom
            lseek(fd1, (i-1) * rsz, SEEK_SET);     // seek to record to be moved
            read(fd1, buf, rsz);                   // read the record
            write(fd1, buf, rsz);                  // write the record - next (lower) position
      }
      lseek(fd2, num * rsz, SEEK_SET);             // seek to record in fdb2
      read(fd2, buf, rsz);                         // read the record
      lseek(fd1, rnum * rsz, SEEK_SET);            // seek to hole created in fdb1
      write(fd1, buf, rsz);                        //write the record
      close(fd1);                                  // close both files
      close(fd2);
      return 0;                                    // return 0 – all good
}
```

Test2dd
1. [25] Write a C function with prototype:
int delrec(char *fdb, int num, int rsz);
This function will delete the record num from the file fdb. Since this is a sequential file of records the records after record num must be moved up to fill in the missing record and the file shortened. Furthermore you may assume that the file consist of an integral number of records and that all records have the same size, rsz. A record is merely a consecutive block of bytes of size rsz. The record indexing starts at 0 so record 0 consists of bytes 0 through rsz − 1, record 1 consists of bytes rsz through 2rsz − 1, and so on. The function must both open and close the file and read/writes must be record sized. If successful, delrec() will return a 0 else delrec() will return -1. The function must use low-level I/O except for printing error messages and you may assume that rsz ≤ 1024.

Data Members:
loop variable, file descriptor, file size variable, buffer for record

Outline:
1. open file
2. get file size
3. move records starting at num +1 up (loop from I = num + 1 to fsize/rsz -1)
    1. lseek to i*rsz
    2. read record into buffer
    3. lseek to (I-1)*rsz
    4. write record in buffer
4. ftruncate(fd, fsize-rsz)
5. close file and return 0

You can assume every open, lseek, read, and write needs to be properly error trapped!

```
int delrec(char *fdb, int num, int rsz){
      int i;                                    // loop variable
      int fd;                                   // file descriptor
      int fsize;                                // size of file
      char buf[rsz];                            // buffer for record
      fd = open(fdb, O_RDWR);                   // open the file
      fsize = lseek(fd, 0, SEEK_END);           // get file size
      for(i = num + 1; i < fsize/rsz - 1; i++){ // loop over lower records to keep
            lseek(fd, i*rsz, SEEK_SET);         // seek to recod to keep
            read(fd, buf, rsz);                 // read the record
            lseek(fd, (i - 1)*rsz, SEEK_SET);   // seek one position up
            write(fd, buf, rsz);                // write the record
      }
      ftruncate(fd, fsize - rsz);               // truncate the file
      close(fd);                                // close the file
      return 0;                                 // return 0 - all good
}
```

Test2e had no record problems
Test2ee same as Test2aa
Test2ff same as Test2dd
Test2gg same as Test2bb
Test2hh same as Test2aa

Test2ii
1. [25] Write a C function with prototype:
int extractrec(char *fdb, int num, char *irec, int rsz);
This function will extract record num from the file whose pathname is given by fdb, writing it to the memory location given by irec. Note that the file will be one record shorter. You may assume that both files consist of an integral number of records and that all records have the same size, rsz. A record is merely a consecutive block of bytes of size rsz. The record indexing starts at 0 so record 0 consists of bytes 0 through rsz − 1, record 1 consists of bytes rsz through 2rsz − 1, and so on. The function must both open and close the file and reads/writes must be record sized. If successful, extractrec() will return a 0 else extractrec() will return -1. The function must use low-level I/O except for printing error messages and you may assume that rsz ≤ 1024.

Test2jj
1. [30] Write a C function with prototype:
int extractrecs(char *fdb, int num, int number, char *fout, int rsz);
This function will extract number records starting with record num from the file whose pathname is given by fdb, writing them to the file whose pathname is given by fout. Note that the file fdb will shorter by number records. You may assume that fdb contains the necessary records and the output file, after execution, should only contain the extracted records. You may assume that both files consist of an integral number of records and that all records have the same size, rsz. A record is merely a consecutive block of bytes of size rsz. The record indexing starts at 0 so record 0 consists of bytes 0 through rsz − 1, record 1 consists of bytes rsz through 2rsz − 1, and so on. The function must both open and close the file and reads/writes must be record sized. If successful, extractrecs() will return a 0 else extractrecs() will return -1. The function must use low-level I/O except for printing error messages and you may assume that rsz ≤ 1024.

Test2kk.1
1. [30] Write a C function with prototype:
int extractrecs(char *fdb, int num1, int num2, char *fout, int rsz);
This function will extract a range of records starting with record num1 and ending with record num2 from the file whose pathname is given by fdb, writing the extracted records to the file whose pathname is given by fout. Note that the file fdb will shorter by the number of records extracted. You may assume that fdb contains the necessary records and that num1 <= num2. The output file, after execution, should only contain the extracted records. You may also assume that the file fdb consist of an integral number of records and that all records have the same size, rsz. A record is merely a consecutive block of bytes of size rsz. The record indexing starts at 0 so record 0 consists of bytes 0 through rsz − 1, record 1 consists of bytes rsz through 2rsz − 1, and so on. The function must both open and close the files and reads/writes must be record sized. If successful, extractrecs() will return a 0 else extractrecs() will return -1. The function must use low-level I/O except for printing error messages and you may assume that rsz ≤ 1024.

Test2kk.2
1. [30] Write a C function with prototype:
int insertrecs(char *fdb, int num, char *fin, int rsz);
This function will insert the records from the file fin after record num in the file whose pathname is given by fdb. You may assume that fdb contains the necessary records, that both files consist of an integral number of records and that all records have the same size, rsz. A record is merely a consecutive block of bytes of size rsz. The record indexing starts at 0 so record 0 consists of bytes 0 through rsz − 1, record 1 consists of bytes rsz through 2rsz − 1, and so on. The function must both open and close the files and reads/writes must (NOTE MUST) be record sized. If successful, insertrecs() will return a 0 else insertrecs() will return -1. The function must use low-level I/O except for printing error messages and you may assume that rsz ≤ 4096.

Test2l
1. [25] Write a C program with invocation:
recselect filein fileout rsz rnum1 rnum2 ... rnumk
This program will read the k records rnum1, . . . , rnumk from filein and append the records to fileout. All records in filein and fileout have the same size, rsz. A record is merely a consecutive block of bytes of size rsz. The record indexing starts at 0 so record 0 consists of bytes 0 through rsz - 1, record 1 consists of bytes rsz through 2rsz - 1, . . . The function must both open and close the files. If successful, recselect will return a 0 else recselect will return -1. The program should use low-level I/O except for error messages and you may assume that rsz ≤ 1024. Notice that rnum1, . . . , rnumk need not be in sequential order, 7, 9, 345, 12 is valid. A correct solution that works for any size record will earn extra credit. On error, return -1.

Test2ll same as Test2bb
Test2mm same as Test2cc
Test2nn same as Test2kk.1

Test2oo
1. [30] Write a C function with prototype:
int mvrec(char *fdb, int num1, char *fdb2, int num2, int rsz);
This function will extract record num1 from the file whose pathname is given by fdb, inserting the extracted record in the file whose pathname is given by fdb2. Note that the file fdb will shorter by one record and the file fdb2 will be longer by one record. You may assume that fdb and fdb2 have more records than both num1 and num2. No records in fdb2 should be overwritten, the record should be inserted. You may also assume that both files fdb and fdb2 consist of an integral number of records and that all records have the same size, rsz. A record is merely a consecutive block of bytes of size rsz. The record indexing starts at 0 so record 0 consists of bytes 0 through rsz − 1, record 1 consists of bytes rsz through 2rsz − 1, and so on. The function must both open and close the files and reads/writes must be record sized. If successful, mvrec() will return a 0 else mvrec() will return -1. The function must use low-level I/O except for printing error messages and you may assume that rsz ≤ 1024.