Πληροφορική & Τηλεπικοινωνίες

Κ18 - Υλοποίηση Συστημάτων Βάσεων Δεδομένων

Χειμερινό Εξάμηνο 2017 – 2018

Καθηγητής Ι. Ιωαννίδης
Άσκηση 2 – Παράδοση 03/12/2017

Για την άσκηση αυτή, θα υλοποιήσετε μία μέθοδο προσπέλασης βασισμένη στα B+ δένδρα που υποστηρίζει τη διεπαφή που περιγράφεται παρακάτω. Θα οργανώσετε ένα αρχείο δύο πεδίων σε B+ δένδρα, χρησιμοποιώντας το πρώτο πεδίο του αρχείου ως το πεδίο-κλειδί. Κάθε αρχείο που αντιστοιχεί σε ένα B+ δένδρο θα αποτελείται από block, τα οποία θα οργανώσετε εσείς κατάλληλα ώστε να έχουν την απαιτούμενη δομή για να λειτουργήσουν ως τέτοια. Κάθε block του αρχείου έχει μέγεθος BLOCKSIZE = 512 bytes.

Συναρτήσεις Επιπέδου Ευρετηρίου AM (Access Method)

void AM_Init()

Η ρουτίνα αυτή χρησιμοποιείται για να αρχικοποιήσετε τις όποιες καθολικές (global) εσωτερικές δομές δεδομένων αποφασίσετε ότι χρειάζεστε να έχετε για την χρήση του B+ δένδρου. Δεν έχει καμμία παράμετρο εισόδου και δεν παράγει καμμία έξοδο.

int AM_CreateIndex(

```
    char *fileName,   /* όνομα αρχείου */
    char attrType1,   /* τύπος πρώτου πεδίου: 'c' (συμβολοσειρά), 'i' (ακέραιος), 'f' (πραγματικός) */
    int  attrLength1, /* μήκος πρώτου πεδίου: 4 γιά 'i' ή 'f', 1-255 γιά 'c' */
    char attrType2,   /* τύπος δεύτερου πεδίου: 'c' (συμβολοσειρά), 'i' (ακέραιος), 'f' (πραγματικός)
    int  attrLength2  /* μήκος δεύτερου πεδίου: 4 γιά 'i' ή 'f', 1-255 γιά 'c' */
```

)

Η ρουτίνα αυτή δημιουργεί ένα αρχείο με όνομα fileName, βασισμένο σε ευρετήριο B+ δένδρου. Το αρχείο δεν πρέπει να υπάρχει ήδη. Ο τύπος και το μήκος του πρώτου πεδίου (αυτού που χρησιμοποιείται για την εισαγωγή στο B+ δένδρο ως κλειδί) περιγράφονται από την δεύτερη και την τρίτη παράμετρο, αντίστοιχα. Παρόμοια, ο τύπος και το μήκος του δεύτερου πεδίου περιγράφονται

απότην τέταρτη και την πέμπτη παράμετρο, αντίστοιχα. Η ρουτίνα επιστρέφει AME_OK εάν επιτύχει, αλλιώς κάποιον κωδικό λάθους.

int AM_DestroyIndex(
    char *fileName /* *όνομα αρχείου* */
)

Η ρουτίνα αυτή καταστρέφει το αρχείο με όνομα fileName, διαγράφοντας το φυσικό αρχείο από το δίσκο. Ένα αρχείο δε μπορεί να διαγραφεί αν υπάρχουν ενεργά ανοίγματα σε αυτό (δείτε στη συνέχεια). Επιστρέφει AME_OK εάν επιτύχει, αλλιώς κάποιον κωδικό λάθους.

int AM_OpenIndex (
    char *fileName /* όνομα αρχείου */
)

Η ρουτίνα αυτή ανοίγει το αρχείο με όνομα fileName. Εάν το αρχείο ανοιχτεί κανονικά, η ρουτίνα επιστρέφει έναν μικρό, μη αρνητικό ακέραιο, ο οποίος χρησιμοποιείται για να αναγνωρίζεται το αρχείο (όπως περιγράφουμε παρακάτω). Σε διαφορετική περίπτωση, επιστρέφει κάποιον κωδικό λάθους.

Θα πρέπει να κρατάτε στην μνήμη έναν πίνακα για όλα τα ανοιχτά αρχεία. Ο ακέραιος που επιστρέφει η AM_OpenIndex είναι η θέση του πίνακα που αντιστοιχεί στο αρχείο που μόλις ανοίχτηκε. Σ' αυτόν τον πίνακα θα κρατάτε ο,τιδήποτε σχετικό κρίνετε ότι πρέπει να είναι άμεσα διαθέσιμο για κάθε ανοιχτό αρχείο (π.χ., γενικές πληροφορίες για το αρχείο, το αναγνωριστικό του αρχείου έτσι όπως έχει ανοιχθεί από το λειτουργικό σύστημα, κτλ. - χωρίς τα παραπάνω να θεωρούνται απαραίτητα). Το ίδιο αρχείο μπορεί να ανοιχτεί πολλές φορές και για κάθε άνοιγμα καταλαμβάνει διαφορετική θέση στον πίνακα που κρατάτε στη μνήμη. Μπορείτε να υποθέσετε ότι οποιαδήποτε στιγμή δεν θα υπάρχουν περισσότερα από MAXOPENFILES = 20 ανοιχτά αρχεία.

int AM_CloseIndex (
    int fileDesc /* αριθμός που αντιστοιχεί στο ανοιχτό αρχείο */
)

Η ρουτίνα αυτή κλείνει το αρχείο που υποδεικνύεται από την παράμετρό της. Επίσης σβήνει την καταχώρηση που αντιστοιχεί στο αρχείο αυτό στον πίνακα ανοιχτών αρχείων. Για να κλείσει επιτυχώς το αρχείο που υποδυκνύεται απο το fileDesc, θα πρέπει να μην υπάρχουν ανοιχτές σαρώσεις σε αυτό. Η συνάρτηση επιστρέφει AME_OK εάν το αρχείο κλείσει επιτυχώς, αλλιώς κάποιον κωδικό λάθους.

int AM_InsertEntry(

```
int fileDesc,   /* αριθμός που αντιστοιχεί στο ανοιχτό αρχείο */
void *value1, /* τιμή του πεδίου-κλειδιού προς εισαγωγή */
void *value2  /* τιμή του δεύτερου πεδίου της εγγραφής προς εισαγωγή */
```

)

Η ρουτίνα αυτή εισάγει το ζευγάρι (value1, value2) στο αρχείο που υποδεικνύεται από την παράμετρο fileDesc. Η παράμετρος value1 δείχνει στην τιμή του πεδίου-κλειδιού που εισάγεται στο αρχείο και η παράμετρος value2 αντιπροσωπεύει το άλλο πεδίο της εγγραφής. Η ρουτίνα επιστρέφει AME_OK εάν επιτύχει, αλλιώς κάποιον κωδικό λάθους.

int AM_OpenIndexScan(

```
int fileDesc, /* αριθμός που αντιστοιχεί στο ανοιχτό αρχείο */
int op,       /* τελεστής σύγκρισης */
void *value  /* τιμή του πεδίου-κλειδιού προς σύγκριση */
```

)

Η ρουτίνα αυτή ανοίγει μία σάρωση (αναζήτηση) του αρχείου που υποδεικνύεται από την παράμετρο fileDesc. Η σάρωση έχει σκοπό να βρει τις εγγραφές των οποίων οι τιμές στο πεδίο-κλειδί του αρχείου ικανοποιούν τον τελεστή σύγκρισης op σε σχέση με την τιμή που δείχνει η παράμετρος value. Οι διάφοροι τελεστές σύγκρισης κωδικοποιούνται ως εξής:

- 1 EQUAL (πεδίο-κλειδί == τιμή της value)
- 2 NOT EQUAL (πεδίο-κλειδί != τιμή της value)
- 3 LESS THAN (πεδίο-κλειδί < τιμή της value)
- 4 GREATER THAN (πεδίο-κλειδί > τιμή της value)
- 5 LESS THAN or EQUAL (πεδίο-κλειδί <= τιμή της value)
- 6 GREATER THAN or EQUAL (πεδίο-κλειδί >= τιμή της value)

Η ρουτίνα επιστρέφει έναν μη αρνητικό ακέραιο που αντιστοιχεί σε μία θέση κάποιου πίνακα που πρέπει να υλοποιήσετε και να κρατάτε στη μνήμη ενήμερο σχετικά με όλες τις σαρώσεις αρχείων που είναι ανοιχτές κάθε στιγμή. Πιθανές πληροφορίες που χρειάζεται κανείς να κρατάει για κάθε ανοιχτή σάρωση σ' αυτόν τον πίνακα είναι το αναγνωριστικό της τελευταίας εγγραφής που διαβάστηκε, ο αριθμός που αντιστοιχεί στο ανοιχτό αρχείο που σαρώνεται, κτλ. (Θα πρέπει να σχεδιάσετε το περιεχόμενο κάθε καταχώρησης στον πίνακα αυτό με βάση αυτά που θα δείτε ότι πρέπει να ξέρετε.) Μπορείτε να υποθέσετε ότι δεν θα υπάρχουν ποτέ πάνω από MAXSCANS = 20 ταυτόχρονες ανοιχτές σαρώσεις αρχείων. Εάν ο πίνακας σαρώσεων είναι γεμάτος, τότε η ρουτίνα επιστρέφει κάποιον κωδικό λάθους. Αντίστοιχους κωδικούς λαθών επιστρέφετε και σε άλλες περιπτώσεις κακής λειτουργίας.

void *AM_FindNextEntry(

int scanDesc */* αριθμός που αντιστοιχεί στην ανοιχτή σάρωση */*

)

Η ρουτίνα αυτή επιστρέφει την τιμή του δεύτερου πεδίου της επόμενης εγγραφής που ικανοποιεί την συνθήκη που καθορίζεται για την σάρωση που αντιστοιχεί στο scanDesc. Αν δεν υπάρχουν άλλες εγγραφές επιστρέφει NULL και θέτει τη global μεταβλητή AM_errno σε AME_EOF. Αν συμβεί κάποιο σφάλμα, επιστρέφει NULL και θέτει τη global μεταβλητή AM_errno στον κατάλληλο κωδικό λάθους.

int AM_CloseIndexScan(

int scanDesc */* αριθμός που αντιστοιχεί στην ανοιχτή σάρωση */*

)

Η ρουτίνα αυτή τερματίζει μία σάρωση ενός αρχείου και σβήνει την αντίστοιχη καταχώρηση από τον πίνακα ανοιχτών σαρώσεων. Επιστρέφει AME_OK εάν επιτύχει, αλλιώς κάποιον κωδικό λάθους.

void AM_PrintError(

char *errString */* κείμενο για εκτύπωση */*

)

Η ρουτίνα τυπώνει το κείμενο που δείχνει η παράμετρος errString και μετά τυπώνει το μήνυμα που αντιστοιχεί στο τελευταίο σφάλμα που προέκυψε από οποιαδήποτε από τις ρουτίνες του AM επιπέδου. Για τον σκοπό αυτό, η ρουτίνα αυτή χρησιμοποιεί μία καθολική (global) μεταβλητή AM_errno η οποία αποθηκεύει πάντα τον κωδικό του πλέον πρόσφατου σφάλματος. Ο κωδικός αυτός σφάλματος πρέπει πάντα να ενημερώνεται σωστά σε όλες τις άλλες ρουτίνες. Η ρουτίνα αυτή δεν έχει δική της τιμή επιστροφής.

void AM_Close()

Η ρουτίνα αυτή χρησιμοποιείται για να καταστρέψετε τι όποιες δομές είχατε αρχικοποιήσει.

Συναρτήσεις BF (Block File)

Το επίπεδο block (BF) είναι ένας διαχειριστής μνήμης (memory manager) που λειτουργεί σαν κρυφή μνήμη (cache) ανάμεσα στο επίπεδο του δίσκου και της μνήμης. Το επίπεδο block κρατάει block δίσκου στην μνήμη. Κάθε φορά που ζητάμε ένα block δίσκου, το επίπεδο BF πρώτα εξετάζει την περίπτωση να το έχει φέρει ήδη στην μνήμη. Αν το block υπάρχει στην μνήμη τότε δεν το διαβάζει από τον δίσκο, σε αντίθετη περίπτωσή το διαβάζει από τον δίσκο και το τοποθετεί στην μνήμη. Επειδή το επίπεδο BF δεν

έχει άπειρη μνήμη κάποια στιγμή θα χρειαστεί να "πετάξουμε" κάποιο block από την μνήμη και να φέρουμε κάποιο άλλο στην θέση του. Οι πολιτικές που μπορούμε να πετάξουμε ένα block από την μνήμη στο επίπεδο block που σας δίνεται είναι οι *LRU* (Least Recently Used) και *MRU* (Most Recently Used). Στην *LRU* "θυσιάζουμε" το λιγότερο πρόσφατα χρησιμοποιημένο block ενώ στην *MRU* το block που χρησιμοποιήσαμε πιο πρόσφατα.

Στη συνέχεια, περιγράφονται οι συναρτήσεις που αφορούν το επίπεδο από block, πάνω στο οποίο θα βασιστείτε για την υλοποίηση των συναρτήσεων που ζητούνται. Η υλοποίηση των συναρτήσεων αυτών θα δοθεί έτοιμη με τη μορφή βιβλιοθήκης.

Στο αρχείο κεφαλίδας bf.h που σας δίνεται ορίζονται οι πιο κάτω μεταβλητές:

BF_BLOCK_SIZE 512 */* Το μέγεθος ενός block σε bytes */*
BF_BUFFER_SIZE 250 */* Ο μέγιστος αριθμός block που κρατάμε στην μνήμη */*
BF_MAX_OPEN_FILES 100 */* Ο μέγιστος αριθμός ανοικτών αρχείων */*

και enumerations:

enum BF_ErrorCode *{ ... }*

Το BF_ErrorCode είναι ένα enumeration που ορίζει κάποιους κωδικούς λάθους που μπορεί να προκύψουν κατά την διάρκεια της εκτέλεσης των συναρτήσεων του επιπέδου BF.

enum ReplacementAlgorithm { LRU, MRU }

Το ReplacementAlgorithm είναι ένα enumeration που ορίζει τους κωδικούς για τους αλγορίθμους αντικατάστασης (*LRU* ή *MRU*).
Πιο κάτω υπάρχουν τα πρωτότυπα των συναρτήσεων που σχετίζονται με την δομή BF_Block.

typedef struct BF_Block BF_Block;

Το struct BF_Block είναι η βασική δομή που δίνει οντότητα στην έννοια του Block. Το BF_Block έχει τις πιο κάτω λειτουργίες.

void BF_Block_Init(*BF_Block **block /* δομή που προσδιορίζει το Block */)*

Η συνάρτηση *BF_Block_Init* αρχικοποιεί και δεσμεύει την κατάλληλη μνήμη για την δομή *BF_BLOCK*.

void BF_Block_Destroy(*BF_Block **block /* δομή που προσδιορίζει το Block */)*

Η συνάρτηση *BF_Block_Destroy* αποδεσμεύει την μνήμη που καταλαμβάνει η δομή *BF_BLOCK*.

void BF_Block_SetDirty(BF_Block *block /* *δομή που προσδιορίζει το Block* */***)**

Η συνάρτηση BF_Block_SetDirty αλλάζει την κατάσταση του block σε dirty. Αυτό πρακτικά σημαίνει ότι τα δεδομένα του block έχουν αλλαχθεί και το επίπεδο BF, όταν χρειαστεί θα γράψει το block ξανά στον δίσκο. Σε περίπτωση που απλώς διαβάζουμε τα δεδομένα χωρίς να τα αλλάζουμε τότε δεν χρειάζεται να καλέσουμε την συνάρτηση.

char* BF_Block_GetData(const BF_Block *block /* *δομή που προσδιορίζει το Block* */ **)**

Η συνάρτηση BF_Block_GetData επιστρέφει ένα δείκτη στα δεδομένα του Block. Άμα αλλάξουμε τα δεδομένα θα πρέπει να κάνουμε το block dirty με την κλήση της συνάρτησης BF_Block_SetDirty. Σε καμία περίπτωση δεν πρέπει να αποδεσμεύσετε την θέση μνήμης που δείχνει ο δείκτης.

Πιο κάτω υπάρχουν τα πρωτότυπα των συναρτήσεων που σχετίζονται με το επίπεδο Block.

BF_ErrorCode BF_Init*(const ReplacementAlgorithm repl_alg /* πολιτική αντικατάστασης */)*

Με τη συνάρτηση BF_Init πραγματοποιείται η αρχικοποίηση του επιπέδου BF. Μπορούμε να επιλέξουμε ανάμεσα σε δύο πολιτικές αντικατάστασης Block εκείνης της LRU και εκείνης της MRU.

BF_ErrorCode BF_CreateFile(*const char* filename* /* όνομα αρχείου */***)***

Η συνάρτηση BF_CreateFile δημιουργεί ένα αρχείο με όνομα filename το οποίο αποτελείται από blocks. Αν το αρχείο υπάρχει ήδη τότε επιστρέφεται κωδικός λάθους. Σε περίπτωση επιτυχούς εκτέλεσης της συνάρτησης επιστρέφεται BF_OK, ενώ σε περίπτωση αποτυχίας επιστρέφεται κωδικός λάθους. Αν θέλετε να δείτε το είδος του λάθους μπορείτε να καλέσετε τη συνάρτηση BF_PrintError.

BF_ErrorCode BF_OpenFile(
const char filename, /* όνομα αρχείου */*
*int *file_desc /* αναγνωριστικό αρχείου block */);*

Η συνάρτηση BF_OpenFile ανοίγει ένα υπάρχον αρχείο από blocks με όνομα filename και επιστρέφει το αναγνωριστικό του αρχείου στην μεταβλητή file_desc. Σε περίπτωση επιτυχίας επιστρέφεται BF_OK ενώ σε περίπτωση αποτυχίας, επιστρέφεται ένας κωδικός λάθους. Αν θέλετε να δείτε το είδος του λάθους μπορείτε να καλέσετε τη συνάρτηση BF_PrintError.

BF_ErrorCode BF_CloseFile(*int file_desc /* αναγνωριστικό αρχείου block */* ***)***

Η συνάρτηση BF_CloseFile κλείνει το ανοιχτό αρχείο με αναγνωριστικό αριθμό file_desc. Σε περίπτωση επιτυχίας επιστρέφεται BF_OK ενώ σε περίπτωση αποτυχίας, επιστρέφεται ένας κωδικός λάθους. Αν θέλετε να δείτε το είδος του λάθους μπορείτε να καλέσετε τη συνάρτηση BF_PrintError.

BF_ErrorCode BF_GetBlockCounter(
const int file_desc, / αναγνωριστικό αρχείου block */*
*int *blocks_num /* τιμή που επιστρέφεται */)*

Η συνάρτηση Get_BlockCounter δέχεται ως όρισμα τον αναγνωριστικό αριθμό file_desc ενός ανοιχτού αρχείου από block και βρίσκει τον αριθμό των διαθέσιμων blocks του, τον οποίο και επιστρέφει στην μεταβλητή blocks_num. Σε περίπτωση επιτυχίας επιστρέφεται BF_OK ενώ σε περίπτωση αποτυχίας, επιστρέφεται ένας κωδικός λάθους. Αν θέλετε να δείτε το είδος του λάθους μπορείτε να καλέσετε τη συνάρτηση BF_PrintError.

BF_ErrorCode BF_AllocateBlock(
const int file_desc, / αναγνωριστικό αρχείου block */*
*BF_Block *block /* το block που επιστρέφεται */)*

Με τη συνάρτηση BF_AllocateBlock δεσμεύεται ένα καινούριο block για το αρχείο με αναγνωριστικό αριθμό blockFile. Το νέο block δεσμεύεται πάντα στο τέλος του αρχείου, οπότε ο αριθμός του block είναι BF_getBlockCounter(...) - 1. Το block που δεσμεύεται καρφιτσώνεται στην μνήμη (pin) και επιστρέφεται στην μεταβλητή block. Όταν δεν χρειαζόμαστε άλλο αυτό το block τότε πρέπει να ενημερώσουμε το επίπεδο block καλώντας την συνάρτηση BF_UnpinBlock. Σε περίπτωση επιτυχίας της *BF_AllocateBlock* επιστρέφεται BF_OK, ενώ σε περίπτωση αποτυχίας επιστρέφεται ένας κωδικός λάθους. Αν θέλετε να δείτε το είδος του λάθους μπορείτε να καλέσετε τη συνάρτηση BF_PrintError.

BF_ErrorCode BF_GetBlock(
const int file_desc, */* αναγνωριστικό αρχείου block */*
const int block_num, */* αναγνωριστικός αριθμός block */*
BF_Block *block /* το block που επιστρέφεται */**)**

Η συνάρτηση BF_GetBlock βρίσκει το block με αριθμό block_num του ανοιχτού αρχείου file_desc και το επιστρέφει στην μεταβλητή block. Το block που δεσμεύεται καρφιτσώνεται στην μνήμη (pin). Όταν δεν χρειαζόμαστε άλλο αυτό το block τότε πρέπει να ενημερώσουμε τον επίπεδο block καλώντας την συνάρτηση BF_UnpinBlock. Σε περίπτωση επιτυχίας της *BF_GetBlock* επιστρέφεται BF_OK, ενώ σε περίπτωση αποτυχίας επιστρέφεται ένας κωδικός λάθους. Αν θέλετε να δείτε το είδος του λάθους μπορείτε να καλέσετε τη συνάρτηση BF_PrintError.

BF_ErrorCode BF_UnpinBlock(*BF_Block *block /* δομή block που γίνεται unpin */)*

Η συνάρτηση BF_UnpinBlock ξεκαρφιτσώνει το block από το επίπεδο BF το οποίο κάποια στιγμή θα το γράψει στο δίσκο. Σε περίπτωση επιτυχίας επιστρέφεται BF_OK, ενώ σε περίπτωση αποτυχίας επιστρέφεται ένας κωδικός λάθους. Αν θέλετε να δείτε το είδος του λάθους μπορείτε να καλέσετε τη συνάρτηση BF_PrintError.

void BF_PrintError(BF_ErrorCode err /* κωδικός λάθους */ **)**

Η συνάρτηση BF_PrintError βοηθά στην εκτύπωση των σφαλμάτων που δύναται να υπάρξουν με την κλήση συναρτήσεων του επιπέδου αρχείου block. Εκτυπώνεται στο stderr μια περιγραφή του σφάλματος.

BF_ErrorCode **BF_Close**()

Η συνάρτηση BF_Close κλείνει το επίπεδο Block γράφοντας στον δίσκο όποια block είχε στην μνήμη. Επιστρέφει ένα μήνυμα λάθους στην περίπτωση που υπάρχουν Unpin Blocks.

Σχόλια για την Υλοποίηση

Εσείς θα αποφασίσετε πώς να δομήσετε εσωτερικά κάθε μπλοκ (κεφαλίδα, ζευγάρια τιμής-δείκτη, κτλ.), τόσο για τα εσωτερικά μπλοκ όσο και για τα μπλοκ-φύλλα του δένδρου. Τα B+ δένδρα που θα υλοποιήσετε θα πρέπει να καλύπτουν την περίπτωση που πολλά ζευγάρια (κλειδί, δείκτης) να έχουν την ίδια τιμή στο κλειδί τους. Μπορείτε όμως να υποθέσετε ότι, για οποιαδήποτε τιμή, τα ζευγάρια με την τιμή αυτή θα είναι λίγα και θα χωρούν άνετα σε ένα μπλοκ.

Επίσης, θα πρέπει να υλοποιήσετε πλήρως τους αλγορίθμους αναζήτησης και εισαγωγής που υπάρχουν για τα B+ δέντρα, συμπεριλαμβανομένης της διάσπασης κόμβων στη διάρκεια εισαγωγής δεδομένων.

Σχολιασμός, Έλεγχος Σφαλμάτων, και Γενική Μορφοποίηση

Όπως πάντοτε, αναμένεται καλός σχολιασμός του προγράμματος, και εσωτερικός (ανάμεσα στις γραμμές κώδικα) και εξωτερικός (στην αρχή κάθε ρουτίνας). Ένας γενικός κανόνας είναι να σχολιάζετε τα προγράμματά σας σαν να πρόκειται να τα δώσετε σε κάποιον άλλον ο οποίος θα τα επεκτείνει και ο οποίος δεν έχει ιδέα για το τι κάνατε όταν τα γράφατε (και δεν μπορεί ούτε να σας βρει να σας ρωτήσει).

Επίσης, θα πρέπει να ελέγχετε για διάφορα σφάλματα που μπορούν να προκύψουν και να βεβαιωθείτε ότι ο κώδικάς σας τερματίζει ομαλά, με μηνύματα που έχουν νόημα, σε όλες τις εισόδους που ικανοποιούν την παραπάνω περιγραφή.

Αρχεία που σας δίνονται

Στα αρχεία που σας δίνονται θα βρείτε δύο φακέλους που περιέχουν το project που θα πρέπει να επιστρέψετε. Ο ένας περιέχει την βιβλιοθήκη η οποία Το project έχει την πιο κάτω δομή:

- **bin**: Ο κώδικας των εκτελέσιμων που δημιουργούνται
- **build**: Περιέχει όλα τα object files που δημιουργούνται κατά την μεταγλώττιση.
- **include**: Περιέχει όλα τα αρχεία κεφαλίδας που θα έχει το project σας. Θα βρείτε το αρχείο bf.h και AM.h.
- **lib**: Περιέχει όλες τις βιβλιοθήκες που θα χρειαστείτε για το project σας. Θα βρείτε το αρχείο libbf.so που είναι η βλιοθήκη για να καλείτε το επίπεδο BF.
- **src**: Τα αρχεία κώδικα (.c) τις εφαρμογής σας.
- **examples**: Σε αυτό τον φάκελο θα βρείτε δύο αρχεία main (bf_main1.c, bf_main2.c) που χρησιμοποιούν το επίπεδο BF καθώς και τις main (am_main1.c, am_main2.c, am_main3.c) οι οποίες χρησιμοποιούν το επίπεδο AM.

Επίσης σας δίνετε και ένα αρχείο Makefile για να κάνετε αυτόματα compile των κωδικά σας.

Πράγματα που πρέπει να προσέξετε

- Θα πρέπει να καλείτε την συνάρτηση *BF_UnpinBlock* για κάθε BF_Block που δεν χρειάζεστε πλέον. Αν δεν τα κάνετε Unpin τότε ο buffer του επιπέδου BF γεμίζει και δεν μπορεί να "θυσιάσει" κάποιο Block γιατί όλα θα είναι ενεργά.
- Πάντα να καλείτε την συνάρτηση BF_Block_SetDirty για τα Block που αλλάζετε τα δεδομένα τους. Αν δεν τα κάνετε dirt τότε δεν γράφονται στον δίσκο.
- Ο κώδικάς που σας δίνετε στο ./examples/hf_main.c δεν θα πρέπει να αλλαχθεί!

Παράδοση εργασίας

Η εργασία είναι ομαδική, **2 ή 3 ατόμων**.

Προθεσμία παράδοσης: 03/12/2017.

Γλώσσα υλοποίησης: C / C++ χωρίς χρήση βιβλιοθηκών που υπάρχουν μόνο στην C++.

Περιβάλλον υλοποίησης: Linux (gcc 5.4+).

Παραδοτέα: Όλος ο φάκελος heap_file_32 ή heap_file_64 καθώς και readme αρχείο με περιγραφή / σχόλια πάνω στην υλοποίησή σας.