1 to 2 – Magnetic Disks — 4 points

Consider a disk drive with the following characteristics:

```
Number of bytes per sector = 512
Number of sectors per track = 100
Number of tracks per cylinder = 8
Number of cylinders = 100
```

and a file containing 8,000 records, each record having 256 bytes.

1 Disks - Part I

How many cylinders would be required to save this file on disk, assuming that the disk is originally empty and an optimal use of space can be achieved?

A. 5

B. 10

C. 15

D. 20

E. 40

2 Disks - Part II

Assume the same organization as in the previous question, and that records were written sequentially into an empty disk in such a way that the tracks of each cylinder were filled before moving to a new cylinder.
How many disk "seeks" are necessary to retrieve the 5th, 50th, 500th, 2500th, 2800th and 3000th record, in this order?

A. 2

B. 3

C. 4

D. 5

E. 6

3 Magnetic Tapes — 2 points

Given a magnetic tape with the following characteristics:

- density: 2000 bpi
- interbloc gap: 0.2 in
- speed = 100 ips

Assume that we want to store a file of 10,000,000 records, using a blocking factor of 1,000. Each record has 100 bytes.
How much space will be necessary to store this file fully?
What is the nominal transmission rate of the tape?

A. Space: 700 inches; nominal transmission rate: 400,000 bytes/sec

B. Space: 5,020 inches; nominal transmission rate: 20,000,000 bytes/sec

C. Space: 502,000 inches; nominal transmission rate: 400,000 bytes/sec

D. Space: 50,200 inches; nominal transmission rate: 200,000 bytes/sec

E. Space: 502,000 inches; nominal transmission rate: 200,000 bytes/sec

4 Buffering — 2 points

Consider the buffering strategies seen in class.
Which one of the following statements is **false**?

A. **Buffer pooling** involves a pool of available buffers from which buffers are taken as needed.

B. **Move mode** involves moving data between an Input/Output buffer and a program buffer.

C. **Locate mode** eliminates the need for transferring data between an Input/Output buffer and a program buffer.

D. When **double buffering** is used, two buffers are involved, and the system operates on one buffer while the other is being loaded or emptied.

E. In both **move mode** and **locate mode**, a program is able to operate directly on data in the Input/Output buffer.

5 Huffman Compression — 2 points

There are several valid Huffman trees for the same file, when we consider the arbitrary choices of right and left subtrees. Each of these choices gives a different Huffman code.

Mark below the answer that **DOES NOT** correspond to a valid Huffman code for the file containing characters: `aabbbc`

A. `a:10, b:0, c:11`

B. `a:00, b:1, c:01`

C. `a:11, b:0, c:10`

D. `a:10, b:11, c:0`

E. `a:01, b:1, c:00`

6 Hashing: predicting record distribution — 2 points

Let r be the number of keys to be stored in a hash table with N addresses.

Let $p(i)$ be the probability that a given address have i keys assigned to it.

The following table shows $p(i)$ approximated by using the Poisson distribution for various values of r/N:

r/N	0.50	0.75	1.00	1.50
p(0)	0.61	0.48	0.37	0.22
p(1)	0.30	0.35	0.37	0.33
p(2)	0.08	0.13	0.18	0.25
p(3)	0.01	0.03	0.06	0.13
p(4)	-	0.01	0.02	0.07

Consider the following two scenarios:

(1) $r = 1000$, $N = 2000$, no buckets

(2) $r = 1000$, $N = 1000$, bucket size 2

Calculate the **expected number of overflow records** (expected number of records to be stored away from their home addresses), for each of the two scenarios respectively:

A. (1) 100 (2) 75

B. (1) 180 (2) 80

C. (1) 180 (2) 360

D. (1) 200 (2) 100

E. (1) 200 (2) 150

7 to 8 - Reclaiming Space in Files — 2 points

Consider the following file that stores information about presidents of Canadian universities; e.g., Edwards is the president of UofO. The file is stored in **variable-length** format with a byte count (length indicator) and 5 variable-length records:

```
         LH ---> -1
Record#: 0              1            2            3                 4
         13Edwards|UofO|11Bates|UofT|11Wills|UofA|16Masteriano|UofW|12Chavez|UofM|
```

Here, `LH` is a pointer to the head of the `AVAIL LIST`. The value -1 is the end-of-list marker. We use a vertical bar as field separator. Two bytes at the beginning of each record store a length indicator; for example, record 0 has a length of 13 bytes, not counting the length indicator itself. We count byte offsets starting with 0; for example, the letter "U" in record 1 has byte offset 23. We use a star (*) to mark deleted records and we organise the `AVAIL LIST` as seen in class.

7 Reclaiming Space in Files - Part I

In the question below, periods show discarded characters; * is put in the first field of the deleted record; and the byte offset of the next available record is put in the second field.

The following is our file after deletion of records number 1 (Bates) and 3 (Masteriano):

A.
```
LH ---> 41
0              1            2            3                 4
13Edwards|UofO|11*.....|-1..11Wills|UofA|16*.........|15..|12Chavez|UofM|
```

B.
```
LH ---> 3
0              1            2            3                 4
13Edwards|UofO|11*.....|-1..11Wills|UofA|16*.........|15..|12Chavez|UofM|
```

C.
```
LH ---> 41
0              1            2            3                 4
13Edwards|UofO|11*.....|15..11Wills|UofA|16*.........|-1..|12Chavez|UofM|
```

D.
```
LH ---> 15
0              1            2            3                 4
13Edwards|UofO|11*.....|-1..11Wills|UofA|16*.........|15..|12Chavez|UofM|
```

E.
```
LH ---> -1
0              1            2            3                 4
13Edwards|UofO|11*.....|41..11Wills|UofA|16*.........|15..|12Chavez|UofM|
```

... continued

8 Reclaiming Space in Files - Part II

Assume the same file organization as in the **previous** question. Which one of the following statements is **false**?

A. We cannot use Relative Record Numbers instead of byte offsets for accessing records directly.

B. When inserting a new record, the first entry (the one `LH` is pointing to) in the avail list must be used.

C. Worst Fit may lead to some amount of fragmentation.

D. We can use the Best Fit replacement strategy for the file.

E. We can use the Worst Fit replacement strategy.

9 Cosequential Processing — 2 points

How many **key comparisons** and **write operations** are required in order to **match** names in the two following lists, assuming that input lists and the output list are to be kept in files on disk:

```
List1:  Banana, Eliza, Finamo, Kant, Ludwig, Pietro, Zaraeu
List2:  Anastasios, Balancius, Furlan, Zoro
```

A. number of key comparisons = 28 number of write operations = 10

B. number of key comparisons = 10 number of write operations = 11

C. number of key comparisons = 10 number of write operations = 0

D. number of key comparisons = 28 number of write operations = 7

E. number of key comparisons = 6 number of write operations = 0

... continued

10 to 11 - Sorting Large Files — 4 points

Given

- a file with 1,000,000,000 records, each record having 1000 bytes,
- 500,000,000 bytes of available memory.

Assume that you would like to sort this file using a merging procedure that divides the file into different runs and merges these runs co-sequentially.

10 Sorting Large Files – Part I

How many runs do you need to divide the file into?

A. 20,000

B. 2

C. 10,000

D. 200

E. 2,000

11 Sorting Large Files – Part II

How many records **in total** can be held simultaneously in memory during the merging part of the algorithm?

A. 50,000

B. 500,000

C. 500,000,000

D. 1,000

E. None of the answers above is correct.

... continued

12 B tree properties — 2 points

Consider an index file of 2,000,000 keys. What is the maximum number of levels that a B-tree index of order 20 has?

A. 7

B. 8

C. 2

D. 100

E. None of the above

13 B+ tree properties — 2 points

Which one of the following assertions about B+ trees is false.

A. A B+ tree consists of an index set and a sequence set.

B. A B+ tree consists of an index set organised as a B-tree.

C. A simple prefix B+ tree uses shortest separators.

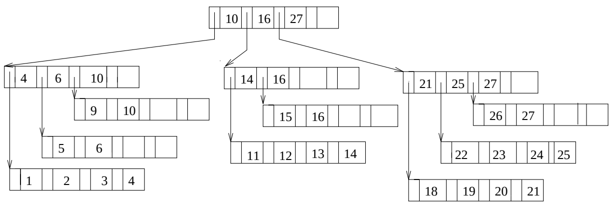
D. Separators are derived from the keys of the records on either side of a block boundary in the sequence set.

E. The index set of a simple prefix B+ tree is a tree over a set of separators.

... continued

14 B tree deletions — 2 points

Consider the following B tree of order 4:



Consider the deletion of keys 9, 16, and 1 in this order.

Determine whether the following statements are TRUE or FALSE:

(1) After 9 is deleted, two nodes were merged.

(2) After 16 and 1 are deleted, the structure of the tree changes.

(3) After 9, 16, and 1 are deleted in this order, the tree still contains 3 levels.

Which of the statements above are TRUE?

A. (1) only

B. (2) only

C. (1) and (2) only

D. all of them are true

E. (1) and (3) only

... continued

15 Hashing — 8 points

Consider the following pairs of key/hash values:

key	A	D	B	E	G	H	F
hash value (home address)	7	6	7	5	4	4	4

In the following four parts you are going to display the hash table resulting from the insertion of the keys above in the given order.
Each hashtable has 8 addresses (the first address is 0 and the last address is 7).
In each part, a different hashing organization is requested.
We draw the table for you only in **Part A**.

Part A — 2 points Hashing: Progressive Overflow
In this part, use progressive overflow (linear probing) and no buckets:

0	
1	
2	
3	
4	
5	
6	
7	

Part B — 2 points Hashing: Progressive Overflow, Buckets
In this part, use progressive overflow (linear probing) and **buckets of size 2**.
The number of hash addresses is still 8 (the first address is 0 and the last address is 7).

... continued

Part C — 2 points Hashing: Chained Progressive Overflow
In this part, use chained progressive overflow with no buckets.

Part D — 2 points Hashing: Scatter Tables
In this part, use a scatter table.

... continued

16 Extendible Hashing: Insertions — 8 points

Perform the following insertions (in the order given below) to an originally empty extendible hashing table:

key: k	$h(k)$	binary representation of $h(k)$ (before bit reversion)
A	0	0000
D	2	0010
B	1	0001
E	5	0101
G	1	0001
H	1	0001
F	5	0101
K	6	0110
L	2	0010

You must use bucket size 3.

You may use the back of the pages as draft.
Show below the extendible hashing structure after each sequence of **3** insertions, namely: after B, after H and after L.

1) Extendible Hashing Structure after inserting A, D, B:

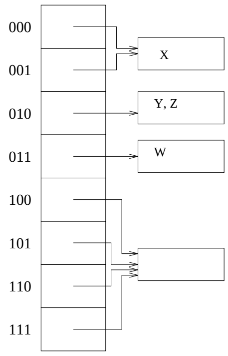
... continued

2) Extendible Hashing Structure after inserting E, G, H:

3) Extendible Hashing Structure after inserting F, K, L:

17 Extendible Hashing: Deletions — 8 points

Consider the following extendible hashing table:



You must use bucket size 2.

Draw the extendible hashing table after each of the following deletions:

A. Delete Z

B. Delete W

For each deletion, **show only the final step as well as any intermediate step where the directory size changes.**

18 B Trees: Insertions — 8 points

Consider the following key sequence:

`3, 19, 4, 20, 1, 13, 16, 6, 2, 23, 14, 7, 21, 18`

Construct a B-tree of order 4, by successive insertion of the keys in the given order.

You may use the back of some pages as draft, and you only need to show below the following intermediate steps:

- an insertion step (of your choice) that causes a simple split;
- an insertion step (of your choice) that causes a recursive split that causes the root to split; and
- the final tree obtained.

1) Selected Simple Split Step:

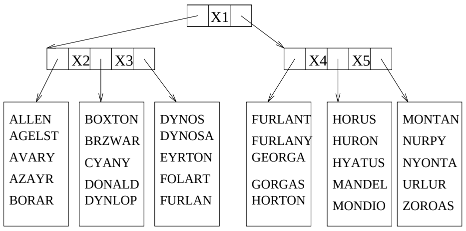
2) Selected recursive split step, spliting the root:

3) Final B tree:

19 Simple Prefix B+ trees — 4 points

Consider the following simple prefix B+ tree structure, storing keys for (a portion of) students of CSI at SITE.

The sequence set contains the keys in blocks of size 5. The B+ tree index set is of order 3.



Fill in the values of the shortest separators that are not revealed, but are represented by unknown symbols:

X1=

X2=

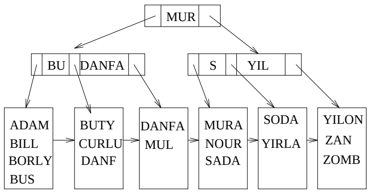
X3=

X4=

X5=

20 B+ tree deletions — 8 points

Consider the simple prefix B+ tree given below with index set of **order 3** and a sequence set with **block size of 4**.



In this question, you must always **give priority to merging** over redistribution.

Use the space below to show how the B+ tree changes after deleting `DANFA, SODA, MURA, NOUR, ZAN, ZOMB, BORLY`, in this order.

You need only to show the tree right after the deletion of `NOUR` and `BORLY`.

B+ tree after the deletion of `NOUR`:

B+ tree after the deletion of `BORLY`:

21 Secondary Indexes: Inverted Lists — 6 points

Suppose that we want to manage the following collection of student records:

```
2087743 Jenkins   Francesca Ottawa
2251841 Duran     Diana     Toronto
2173098 McKone    Ryan      Toronto
2240890 Adelstein Katharine Quebec
2180384 Choi      Maria     Shawinigan
2231301 Hemmings  Anne      Montreal
2225639 Taylor    Sarah     Toronto
2225657 Jeppesen  Abigale   Ottawa
```

The fields in the datafile above contain: identification number (studId), last name, first name, and city of origin, respectively.

Below are a primary index file, and secondary index file organized using inverted lists:

Primary Index

	primary key	reference to datafile
0	2087743	0
1	2173098	3
2	2180384	4
3	2225639	6
4	2225657	7
5	2231301	5
6	2240890	3
7	2251841	2

Secondary Index

	secondary key	ref to Id List
0	Montreal	5
1	Ottawa	0
2	Quebec	3
3	Shawinigan	4
4	Toronto	2

Id List File

0	2087743	7
1	2251841	2
2	2173098	6
3	2240890	-1
4	2180384	-1
5	2231301	-1
6	2225639	-1
7	2225657	-1

Each entry in **Secondary Index** points to a list of its corresponding primary keys stored in **Id List File**. Notice that each of these lists is sorted.

Show the content of the 3 files after inserting (in this order) the following 3 records into the datafile:

```
2351741 Malik     Anna      Toronto
2246757 Jorg      Derek     Toronto
1987744 Laberge   Jean      Gatineau
```

Contents of the 3 files after the 3 insertions above:

Primary Index

	primary key	reference to datafile
0		
1		
2		
3		
4		
5		
6		
7		
8		
9		
10		
11		

Secondary Index

	secondary key	ref to Id List
0		
1		
2		
3		
4		
5		
6		
7		

Id List File

0		
1		
2		
3		
4		
5		
6		
7		
8		
9		
10		
11		

22 Hashing Insertion Pseudocode — 12 points

Write pseudocode for `inserting` in a hashing file using progressive overflow (or linear probing). The hashing file may contain **tombstones**, that is, a record which has been previously occupied, but it is not occupied anymore.

You may use the following guidelines when writing your pseudocode:

- The hashing record contains two fields: **`key`** field (integer), **`reference`** field (integer). The `key` field contains:

  a positive integer, if it represents a real key;

  `0`, if this position is free;

  `-1`, if position contains a tombstone.

- Your pseudocode must give details about low level file manipulation operations. For example:
  `Open logical file Y in reading mode associating it to physical file X;`
  `Position the reading pointer for file Z at byte offset N;`
  `Read (key,reference) from file W;` (here we assume it is reading from current reading position)
  `Position the writing pointer for file Z at byte offset N;`
  `Write (key,reference) into file W;` (here we assume it is writing into current writing position)
  etc, etc.

- The data to be inserted into the hashing file is `(newkey, newreference)`.

A skeleton for your pseudocode is in the next page.


```
######################## PSEUDOCODE: ############################
Global Variables and constants to be used:
   HASHSIZE: number of hash addresses
   "hashindex.txt": physical file containing HASHSIZE records
                    (RRNs from 0 to HASHSIZE-1)
   RECSIZE: number of bytes used for each record of "hashindex.txt"
   hashfile: logical name to be associated to "hashindex.txt"

int HashFunction(int key); // returns the hashing home address for the key

int HashSearch(int key); // returns the RRN of the position of the key in the
                         // hashfile, if the key is present, or -1, otherwise

procedure HashInsert(int newkey, int newreference)
{
   Open hashfile in reading & writing mode, associating it to "hashindex.txt"




    Close hashfile;
}
```


23 Lempel-Ziv Decompression Program — 12 points

Write a C++ program that, given a Lempel-Ziv compressed file, produces the **decompressed** file. PLEASE, READ WELL: YOU MUST DECOMPRESS, not compress.

Recall that the compressed file (input) consists of a sequence of several (number,letter) pairs. You don't need to program the details of how to read these pairs. Indeed, you should use procedure `ReadPairsFromFileToArrays` which reads from the input file the pair into arrays `numbers` and `letters` for you. Use the back of the page if you need more space.

```
#include <fstream>
#include <iostream>
using namespace std;
#define MAXSIZE 32512

void ReadPairsFromFileToArrays(istream & inputfile, int numbers[], char letters[]);
   // assume this procedure is given (use it, no need to write it)

int main() {
   istream input; ostream output; // input and output files
   int numbers[MAXSIZE]; char letters[MAXSIZE]; // arrays with compressed info
   input.open("compressed.txt",ios::in|ios::binary); // open compressed file
   ReadPairsFromFileToArrays(input,numbers,letters); // read pairs from file



}
```