CSCI 352 Unix Software Development
Summer 2015
Assignment 2

Submitting Your Work

This assignment is worth 15% of the grade for the course. Save your program files (including header files) in the zipped tar file WnnnnnnnnAssg2.tar.gz (where Wnnnnnnnn is your WWU W-number) and submit the file via the **Assignment Submission** item on the course web site. You must submit your assignment by 9:00am on Monday, August 3.

Your assignments will be evaluated on correct functionality and conformance to the coding standards described at the end of this assignment specification.

The Task

Your task for this assignment is to implement a data file as a B-tree, thus enabling both efficient random and sequential access to the data items stored in the file. Note that this is not a dynamic data structure stored in heap memory; it is a dynamic data structure stored in a single file in secondary storage.

The Data Items

The data items to be stored in the file contain inventory and sales history data for a retail outlet. Each data item represents one product in the company's catalog. The data items are all of the same length, comprising the following fields:

- Product code: 8 characters.
- Product description: 30 characters (blank padded to the right).
- Product price dollar amount: integer.
- Product price cent amount: integer.
- Product category: 12 characters (blank padded on the right).
- Current stock: integer.
- Sales history: 12 integers, representing sales of the item for each of the past 12 months.

The initial data is provided in the file inventory.txt.

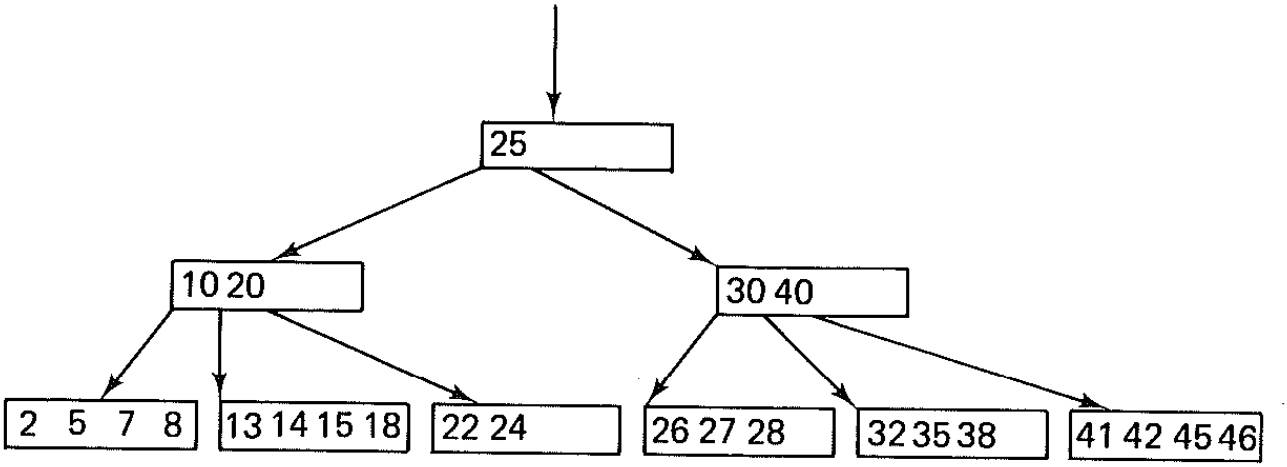
B-Trees

Wikipedia: "**Not to be confused with Binary tree**. A B-tree is a tree data structure that keeps data sorted and allows searches, sequential access, insertions, and deletions in logarithmic time. The B-tree is a generalization of a binary search tree in that a node can have more than two children. Unlike self-balancing binary search trees, the B-tree is optimized for systems that read and write large blocks of data. B-trees are a good example of a data structure for external memory. It is commonly used in databases and file systems".

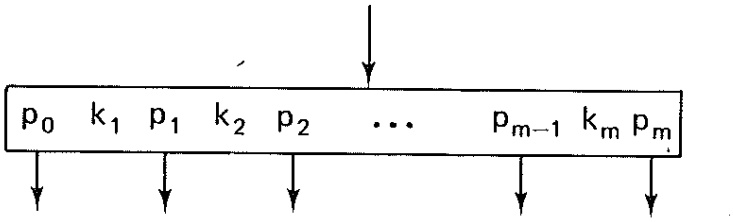
Each node (page) in a B-tree contains several data items. If the *order* of the B-tree is *n*, then each page (apart from the root page) must contain at least *n* data items and at most *2n* data items. The root page may contain zero data items (when the B-tree is empty) up to *2n* data items. B-trees have the following characteristics:

1. Every page contains at most *2n* data items.
2. Every page, except the root page, contains at least *n* data items.
3. Every page is either a leaf page (it has no descendants) or it has *m+1* descendants, where *m* is the number of data items in the page.
4. All leaf pages are at the same level.

The following shows a B-tree of order 2, with integers for data items. The tree has 3 levels.

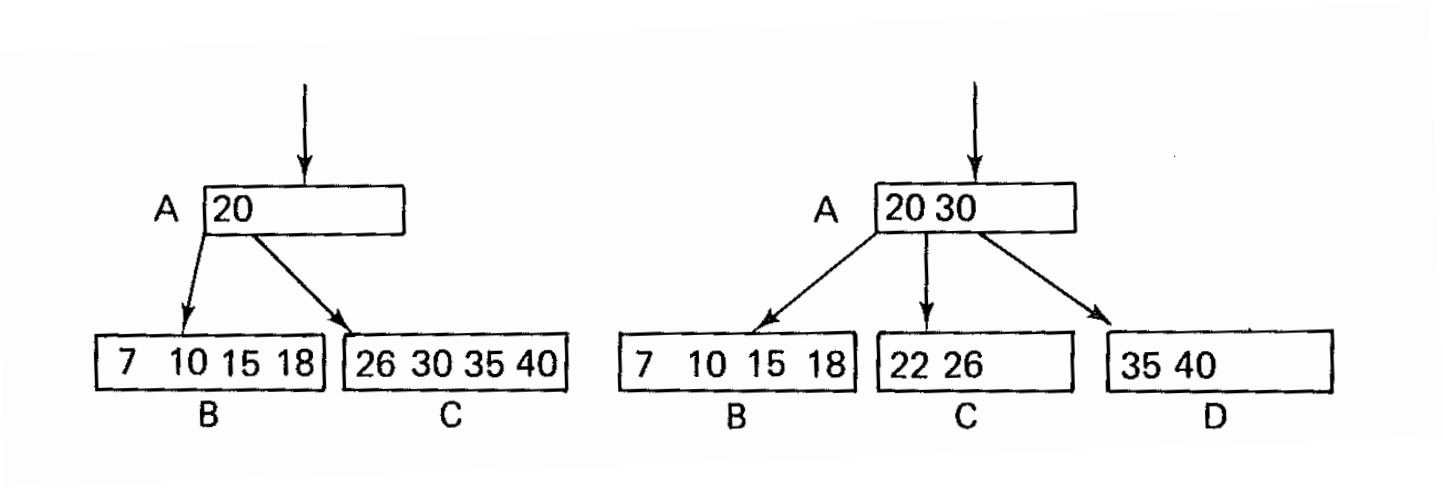


Each page has the form:



Here, $k_1, k_2, \ldots, k_m$ are the keys for the *m* data items and $p_0, p_1, \ldots, p_m$ are "pointers" to the descendant pages.

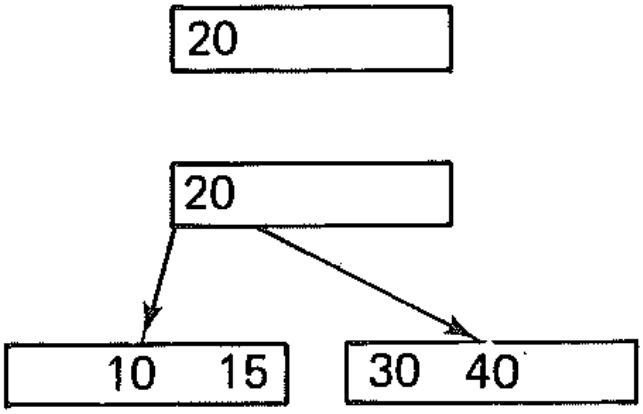
When a new data item is to be added to a page which already has *2n* data items, that page must split into 2 pages and the middle data item is promoted to the parent page. This is shown below, where item 22 is added to a full page in the B-tree on the left. The outcome is shown in the B-tree on the right. Item 22 belongs in page C, but that page is already full. Page C is split evenly into pages C and D and the middle item is promoted

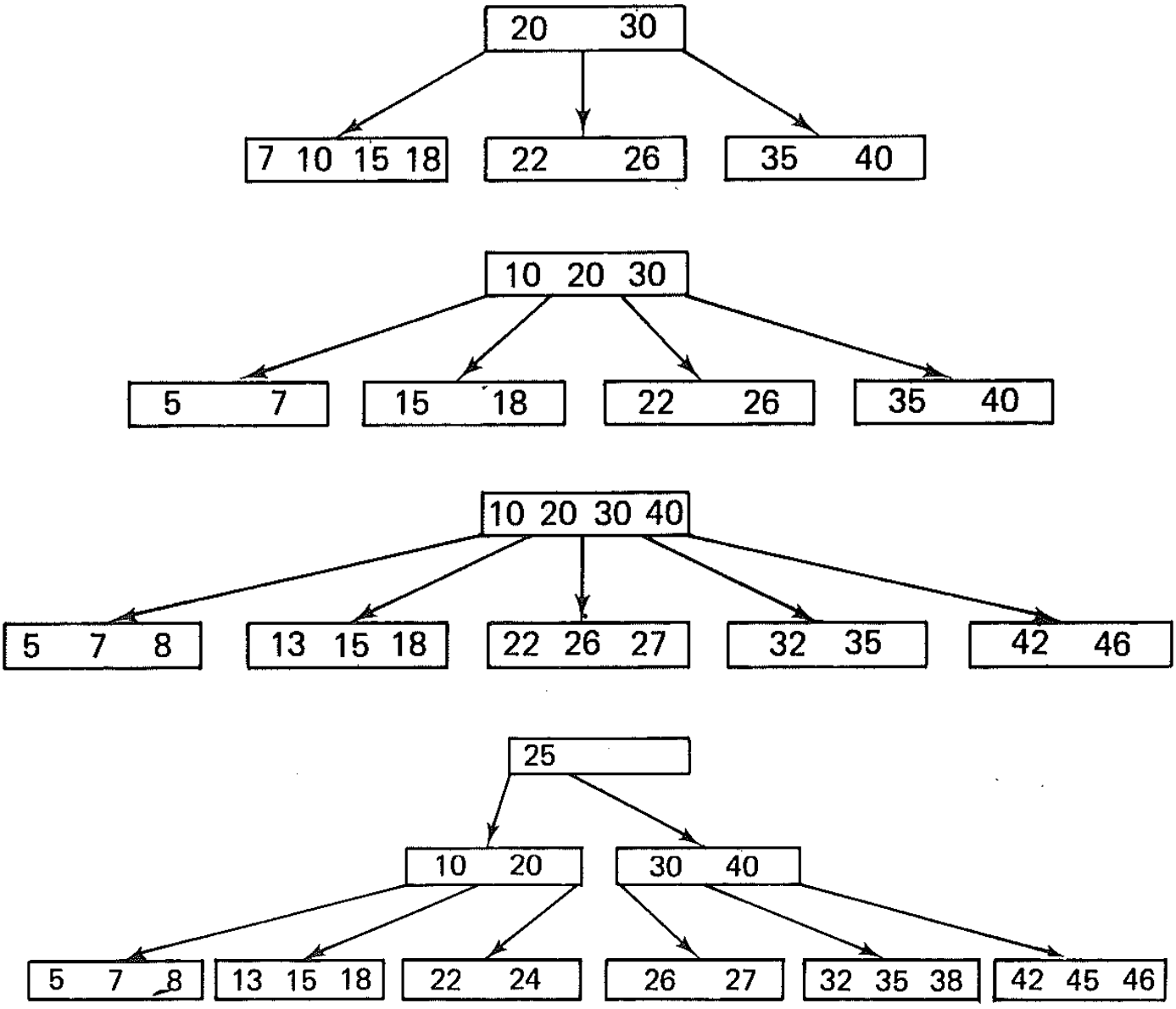


The following sequence of diagrams show the construction of a B-tree with the following insertion of keys:

20; 40 10 30 15; 35 7 26 18 22; 5; 42 13 46 27 8 32; 38 24 45 25;

The semi-colons in the above list indicate places where a snapshot is taken of the tree.





Transactions

Once you have loaded the data from inventory.txt into the B-tree, you need to run another program to apply transactions against the records stored in the B-tree file.

Transactions are events which cause a change to the inventory data. There are five types of transaction in this system:

- Sale: a quantity of one product has been sold and is removed from the stock for that product.
- Delivery: a quantity of one product has been delivered and is added to the stock for that product.
- New product: a new product is added to the inventory.
- Delete product: an existing product is deleted completely from the inventory.

- Price change: the price of one product is changed.

Transaction File

The transactions for the last month are provided in the file `transactions.txt`. Each line in that file gives the data for one transaction. There may be more than one transaction for any product. The data given for a transaction depends on the type of transaction.

- Sale: product code and quantity sold. For example:

  ```
  SALE WL237870 2
  ```

  This represents a sale of 2 units of the product with code WL237870.
- Delivery: product code and quantity delivered. For example:

  ```
  DELIVERY WL242373 10
  ```

  This represents a delivery of 10 units of the product with code WL242373.
- New product: product code, description, price and category of a new product. For example:

  ```
  NEWPRODUCT WL220312New Wave Jacket                  55.50 jacket
  ```

  This represents a new product with product code WL220312, description "New Wave Jacket" (padded out with spaces to 30 characters), price $55.50, and category jacket.
- Delete a product: product code. For example:

  ```
  DELETE WL036147
  ```

  This means that the product with code WL036147 is to be deleted from the inventory.
- Price change: product code and new price. For example:

  ```
  PRICE WL162343 45.50
  ```

  This indicates that the price for the product with code WL162343 is to be changed to $45.50.

The transactions are listed in the file in the order in which they occurred during the last month. Therefore, if there is a price change for a product and two sales transactions for that same product, one before the price change and one after the price change, the first sale will be valued at the old price and the second sale will be valued at the new price.

What your program must do

1. Read each record from the inventory data file inventory.txt and load it into the B-tree file.
2. Read the transactions from the file transaction.txt and apply each transaction to the inventory data held in the B-tree file, updating the B-tree file.
3. For a new product, the sales history is to be set to zero for each month.
4. The total sales quantity for the month is to be added as the first component of the sales history, pushing all the existing history items along one place and discarding the last number from the sequence.
5. Check for errors in the transactions. If any error is found, display a message on standard output with the details of the transaction and the nature of the error. A transaction containing any error is not to be used to update the inventory data.

   The type of error to look for depends on the type of transaction.

   - For sale, delivery, delete and price change transactions, report an error if the transaction product code does not exist in the inventory.

- For a new product, report an error if a product with that code already exists in the inventory.
- For a sale, report an error if the sale quantity is larger than the stock of that product in the inventory.

6. The inventory file is to be updated as part of processing each transaction.
7. After all the transactions have been processed, display a report on standard output showing product code, product description and the total sales value (total dollar amount) for each product for the month represented by the transaction file. The products are to be listed in ascending order of product code.
8. You **must** use a separate source file for functions that manipulate the B-tree file, a separate source file for a program to read the inventory file and initialize the B-tree file, and a separate source file for another program that will apply the transactions to the inventory.
9. The order of the B-Tree must be a constant, defined with a #define in the C source code.

Saving your files in a zipped tar file

You need to submit all your C and header files and a makefile that can be used to compile and link your programs. First you need to bundle all the files up into a single *tar* file. The term "tar" is an abbreviation of "tape archive" and goes back to the days when people would save a back-up copy of their files on magnetic tape. Nowadays, with the price of large disk drives so low, nobody uses tape anymore, but the concept of tar files survives.

1. Use the command:

   ```
   tar -cf WnnnnnnnnAssg1.tar *.c *.h
   ```

   (where Wnnnnnnnn is your W-number).

   The -cf specifies two options for the tar command: 'c' means create and 'f' means that the name of the resulting tar file comes next in the command.

   Following the name of the tar file, we list the files to be included in the tar file. In this case, we want all the files in your current directory whose names end with `".c" or ".h"`.

2. If you now use the command `ls` you should now see the file WnnnnnnnnAssg1.tar in your directory.
3. If you use the command

   ```
   tar -tf WnnnnnnnnAssg1.tar
   ```

   (where Wnnnnnnnn is your W-number), it will list the files within the tar file.

4. Now compress the tar file using the gzip program:

   ```
   gzip WnnnnnnnnAssg1.tar
   ```

5. By using the `ls` command again, you should see the file WnnnnnnnnAssg1.tar.gz in your directory. This is the file that you need to submit through the **Assignment 1 Submission** link in the course web site.

Coding Standards

1. Use meaningful names that give the reader a clue as to the purpose of the thing being named.
2. Avoid the repeated use of numeric constants. For any numeric constants used in your program, use #define preprocessor directives to define a macro name and then use that name wherever the value is needed.

3. Use comments at the start of the program to identify the purpose of the program, the author and the date written.
4. Use functions to provide a solution to each sub-problem. Ensure that no function gets too large. Use 20 lines maximum as a guideline.
5. Use comments at the start of each function to describe the purpose of the function, the purpose of each parameter to the function, and the return value from the function.
6. Use comments at the start of each section of the program to explain what that part of the program does.
7. Use consistent indentation.