Simulation Program to Study MRP Nervousness

by

Adiel G. Smith

and

Luke Shulenburger

SUBMITTED TO THE DEPARTMENT OF
MECHANICAL ENGINEERING IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE OF

BACHELOR OF SCIENCE IN MECHANICAL ENGINEERING

AT THE

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

JUNE 2002

Signature of Author: [illegible]
Department of Mechanical Engineering
May 10, 2002

Certified by: [illegible]
Stanley Gershwin
Senior Research Scientist
Thesis Supervisor

Accepted by: [illegible]
Ernesto Cravalho
Professor of Mechanical Engineering
Chairman, Department of Mechanical Engineering

Simulation Program to Study MRP Nervousness

by

Adiel G. Smith

and

Luke Shulenburger

Submitted to the Department of Mechanical Engineering
on May 10, 2002 in Partial Fulfillment of the
requirements of the Degree of Bachelor of Science in
Mechanical Engineering

ABSTRACT

Material Resource Planning (MRP) is a common production system used by many of today's manufacturing facilities. When interactions between a manufacturer and supplier become complex, MRP systems sometimes become chaotic, an event called MRP Nervousness. MRP Nervousness occurs when small scheduling updates lead to very large changes in the finished product schedule. A program was developed to allow a user to create an MRP system and derive outputs given certain known MRP system variables and rules. This program will be used in the study of MRP Nervousness.

Thesis Supervisor: Stanley Gershwin
Title: Senior Research Scientist

Dedication and Acknowledgement

We would like to thank our families and friends for standing by our side throughout our four years at MIT.

The authors would like to recognize Dr. Stanley Gershwin for providing insight and support and for stimulating interest in the field of Manufacturing.

Table of Contents

List of Tables and Figures

1. Introduction

Material Requirements Planning, known simply as MRP, is a system used in production control. This system is used to determine the optimum time at which production material should be ordered. Developed at IBM in the early 1960s, MRP gained popularity in the 1970s when it was supported by the American Production and Inventory Control Society (APICS). Since APICS's rallying cry, MRP has become the most common production control method in the United States.

Joseph Orlicky, one of MRP's creators in the 1960s, realized that the current techniques of inventory management and production control were insufficient. In most of those systems, part production started when its inventory dropped below a certain level. This system did not differentiate between the final product and components of that final product.

The developers of MRP realized that there are actually two types of demand in a manufacturing system: *independent* and *dependent.* Independent demand comes from outside orders for the final product. This demand is, by nature, unpredictable. However, dependent demand is stable. It is defined as component demand, necessary to final product completion. The MRP system recognizes this difference: "Production to meet dependent demand should be scheduled so as to explicitly recognize its linkage to production to meet independent demand" (Hopp and Spearman, 106).

MRP is based on the idea that once a product's final due date is known, if all of the production variables are known (such as lead time, production time, and order time), due dates for each component can be set. The MRP system links independent and dependent demand. The system calculates when parts should be started given the demand for the final product (Hopp & Spearman, 1996).

2. MRP

2.1 Bill of Materials

The Bill of Materials (BOM) represents the process of producing a finished product. A simple BOM is shown in Figure 1:

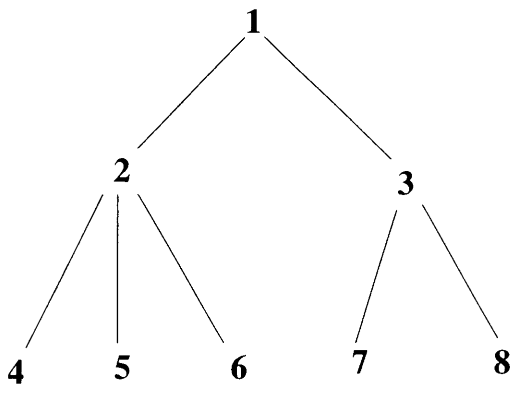

Figure 1: Example Bill of Materials

A fork in the tree shows what subparts make up a part. In this example, part 2 is made up of subparts 4, 5, and 6. The MRP program developed allows a user to create a BOM and assign properties such as inventory and production time to each part.

2.2 Batching

Batching is the term describing ordering rules between the manufacturer and the supplier. One common batching rule (used in this MRP program) sets lot sizes (amount of items per order) constant. This is known as Fixed Order Quantity (FOQ) batching. For example, a car manufacturer might order car tires in constant lot sizes of 100 tires. This means that the minimum order of tires the manufacturing plant can order is 100 tires. Other rules include lot-for-lot ordering, where one will order the exact amount of parts needed for a given time period.

2.3 MRP Example

To help readers understand the MRP system, a simple example will demonstrate the basic calculations.

Imagine that final product F needs to be delivered to a customer in 3 weeks[1]. Item F is made from assemblies A and B. Item A takes 1 week to build (this is defined as lag time) and is used twice in final product F. Item B takes 2 weeks to build and is only used once in final product F. Using basic logic, we know that the MRP schedule will look like this:

[1] For this example, the unit of time (or time unit) is weeks. This assumes that the production/ordering schedule is segmented by weeks. The unit of time can also be segmented by hours, days, months, or any other time unit.

Table 1: Simple MRP Production Schedule

	Week 1	Week 2	Week 3
Item F			1
Item A		2	
Item B	1		

2.4 MRP Algorithm

There are several key relations that are used in any standard MRP program. The key variables in this MRP program are: lot size, lead time, on-hand inventory, and demand.

Demand, D_t, is defined as the demand for the time unit t. Inventory for the end of any period t is expressed as I_t. A main set of equations relating inventory to demand is given as

$$I_t=I_{t-1} - D_t \text{ if } I_{t-1} - D_t \geq 0 \tag{1}$$

and

$$I_t=I_{t-1} - D_t + S_t \text{ if } I_{t-1} - D_t < 0 \tag{2}$$

where S_t is the quantity that will be completed in time unit t.

Equations 1 and 2 represent the main algorithm upon which additional variables are added. Lead time is a good example of how the MRP system gets more complex. If a subpart takes two weeks to complete (or order), then it is logical that it should be started (or ordered) two weeks prior to the deadline. The lead time in this case is two weeks. The program must take lead time into account when producing an ordering and production schedule.

This system gets complicated when the same part is used in different parts of the Bill of Materials. Other complications include idiosyncrasies such as the need for modified demand.

Modified demand occurs when it is not possible to produce a final part to meet demand. For example, consider the case from above as seen in Table 1. In this example, the lead time for Item B is two weeks. However, if the lead time for this item were three weeks instead, it would be impossible to produce the final Item F on time. Instead, the part would be ready (yet late) in week four. The demand gets pushed to the next week. This is what is meant by modified demand.

The MRP program is equipped to handle these and other special cases that complicate MRP systems. Close inspection of the Programming Manual in Section 4 will give a more detailed account of the logical flow of the MRP program. Analysis of the program code will also be helpful in understanding the main program algorithm.

2.5 MRP Nervousness

While MRP is a very logical system, a phenomenon called MRP Nervousness exposes some of its weaknesses. MRP Nervousness occurs "when a small change in the master production schedule results in a large change in planned order releases" (131). This phenomenon turns an orderly system into a chaotic one. In a study of MRP Nervousness, David Greenwood (2001) determined that if the manufacturing facility and the suppliers were acting in accordance with MRP guidelines, and chaotic scenarios arose, then the system itself must be at fault. MRP Nervousness can be crippling to a manufacturing plant. Problems that arise include worker confusion, excessive part setups, and the consumption of excess expense and/or capacity. All of these problems add up to increased costs and customer dissatisfaction. This thesis' focus is to provide an easy-to-use program that will enable the study of MRP Nervousness.

3. User Manual

3.1 Input File

The input file of the MRP program allows the user to create a Bill of Materials[2] and assign certain properties and rules for the parts and for the system. This program interacts with the other programs and will produce an output file (as will be described later throughout Section 3.

The program allows the following subpart information to be specified: lot size, lead time, number of subparts needed for assembly, number of subparts needed for assembly, on-hand inventory for week zero, and number of parts needed for assembly. The program also allows specification of the number of total weeks and the demand for each week.

3.2 Output File

When the input file is entered into the system, the program will produce an output file. This file will display all relevant ordering information. This information includes the input demand and the output. The file will also show each part with the production schedule for that part. The file will also display the inventory levels for each part for each time-unit.

3.3 Input File Template

The following example outlines the input template for the MRP Program:

Enter number of time periods:

[2] For simplicity, assume that the time unit used in the following examples is weeks.

*

Next, enter the final product demand for each time period

*
*
*...

Starting at the top of the BOM tree, enter information in the following order: Part number, lot size, lead time, number of subparts, and on-hand inventory for week zero. Next, enter each subpart number followed by the number of this subpart required for this assembly. This should be done on the same line leaving a space between numbers.

*
*
*
*
*

Enter the data for the tree from top to bottom, staying as far to the left as possible without returning to parts already visited. When the bottom of a branch is reached, follow the previous instructions, but do not fill in anything for the subparts. After reaching the bottom of a branch, traverse up to the nearest node whose subparts have not been defined and then traverse down those (staying as far to the left as possible). If there is a repeat part in the Bill of Materials, only enter that part number. Do not fill out any information about lot size, lead time, etc... When the whole Bill of Materials has been defined, enter a (-1) to let the program know that the Bill of Materials is completely defined[3].

3.4 Input File Example Description

Below is a picture of the BOM and the format for BOM entry. This example will demonstrate how to create an input file for any Bill of Materials. The parts are numbered in the order that they should be defined. As mentioned before, it is necessary to start at the top and work downward, staying as far to the left as possible.

[3] This program can handle part numbers from 1 to 65,536.

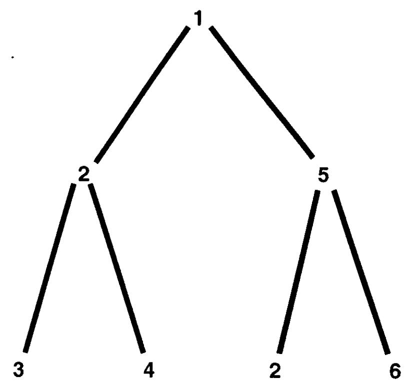

Figure 2: Example Bill of Materials

Assume that we are looking at 5 time periods, with the demand set as:

Week 1:	5
Week 2:	5
Week 3:	30
Week 4:	20
Week 5:	10

Assume all parts have fixed lot size of 10, lead time of 1.

3.5 Example Input File[4]:

5	*number of time periods*
5	*demand for part 1, week 1*
5	*demand for part 1, week 2*
30	*demand for part 1, week 3*
20	*demand for part 1, week 4*
10	*demand for part 1, week 5*
1	*part 1*
10	*lot size for part 1*
1	*lead time part 1*
2	*number of subparts for part 1*
20	*inventory for part 1*
2 1	*first subpart, only need one part per assembly*

[4] Note: It is not necessary to leave spaces between lines. However, this method is suggested to assist in editing. The comments for the input file are italicized.

5 1	*second subpart, only need one part per assembly*
2	*part 2*
10	*lot size for part 2*
1	*lead time part 2*
2	*number of subparts for part 2*
80	*inventory for part 2*
3 1	*first subpart, only need one part per assembly*
4 1	*second subpart, only need one part per assembly*
3	*part 3*
10	*lot size part 3*
1	*lead time part 3*
0	*# subparts for part 3 (0 indicates the end of a branch)*
10	*inventory part 3*
4	*part 4*
10	*lot size part 4*
1	*lead time part 4*
0	*# subparts for part 4 (0 indicates the end of the branch)*
50	*inventory part 4*
5	*part 5*
10	*lot size part 5*
1	*lead time part 5*
2	*# subparts for part 5 (0 indicates the end of the branch)*
30	*inventory part 5*
2 1	*first subpart, only need one part per assembly*
6 2	*second subpart, only need one part per assembly*
2	*part 6—a repeat only requires the part number*
6	*part 6*
10	*lot size part 6*
1	*lead time part*
0	*# subparts for part 6 (0 indicates the end of the branch)*
10	*inventory part 6*
-1	*Indicates end of program*

3.6 Output File Example Description

An output file in generated after the input file is run through the computer program. The program will display all relevant information such as: final product weekly output, part weekly inventory and ordering times. The output file is easy to decipher. The following output file from the above example shows what a typical output file will look like.

3.7 Example Output File:

```
Below is a list of how many of the final part were output each week

Week 1   output = 5, modified demand[5] = 5, demanded = 5
Week 2   output = 5, modified demand = 5, demanded = 5
Week 3   output = 30, modified demand = 30, demanded = 30
Week 4   output = 20, modified demand = 20, demanded = 20
Week 5   output = 10, modified demand = 10, demanded = 10


Below is a listing of how many of each part we need to start producing each week.

Part Number 1
Week 2:  20
Week 3:  20
Week 4:  10
Part Number 5
Week 2:  10
Week 3:  10
Part Number 6
Week 1:  10
Week 2:  20

Below is a listing of the inventory of each part in each week[6].

Part Number 1, Inventory:
Week 1:   15
Week 2:   10
Week 3:   0
Week 4:   0
Week 5:   0
Part Number 2, Inventory:
Week 1:   80
Week 2:   50
Week 3:   20
Week 4:   10
Week 5:   10
Part Number 3, Inventory:
Week 1:   10
```

[5] Modified demand equals original demand plus any rescheduled demand that had to be pushed back.
[6] Note that this is the inventory at the end of the week

Week 2: 10
Week 3: 10
Week 4: 10
Week 5: 10
Part Number 4, Inventory:
Week 1: 50
Week 2: 50
Week 3: 50
Week 4: 50
Week 5: 50
Part Number 5, Inventory:
Week 1: 30
Week 2: 10
Week 3: 0
Week 4: 0
Week 5: 0
Part Number 6, Inventory:
Week 1: 10
Week 2: 0
Week 3: 0
Week 4: 0
Week 5: 0

3.8 Nervousness Example

The following is an example modified from Greenwood's (2001) paper used to demonstrate Nervousness. The BOM is shown below in Figure 3:

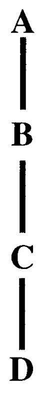

Figure 3: Nervousness BOM

This simple BOM shows that end item A is made from subassembly B. Subassembly B is made of subassembly C, which in turn is made from an ordered part D. Assume for this example that the lot size for B is 5 parts and the lot size for C is 12 parts. There is no lot size for part D. Also assume that the lead time for each part is zero. Table 2 shows the original schedule for the BOM.

Table 2: Original Schedule

	Overdue	Week 1	Week 2	Week 3	Week 4	Week 5	Week 6
A	2	2	2	2	2	2	2
B	5	-	5	-	-	5	-
C	12	-	-	-	-	12	-
D	12	-	-	-	-	12	-

A small change to the demand of the final product A is made in Week 4. The demand for Week 4 is changed to 3 parts, while the demand for Week 5 is reduced to 1. However, these small effects produce a significant scheduling change in the ordering schedule as seen in Table 3.

Table 3: Modified Schedule

	Overdue	Week 1	Week 2	Week 3	Week 4	Week 5	Week 6
A	2	2	2	2	3	1	2
B	5	-	5	-	5	-	-
C	12	-	-	-	12	-	-
D	12	-	-	-	12	-	-

After moving one item in the demand one week earlier, twenty-nine parts had to be produced one week earlier. This example shows how a small change in final product demand can snowball and produce large changes down the line. Most manufacturing plants have much more complicated BOMs and schedules and so these small changes can produce extreme changes.

4 Programming Manual

4.1 Background

There are multiple programs that make up the greater MRP program. The Programming Manual details the structure and interaction of the programs. Although all of the programs are thoroughly commented, the Manual will be a big aid should any modifications be necessary in the future.

The MRP program is written in C. It takes one command line argument that is the name of the file containing the input. The format of this file is specified in the user's manual. It is a plain text file that describes the system to be analyzed. It contains a single entry per line.

4.2 Data Structures

At the heart of the program is a tree (representing the BOM) that holds the information about all of the parts that are used to make the final product. Each element in the tree is a struct of type part that is described in the file *inventory.h.* Each part contains an array of type subpart which allows the part to be connected to the tree. Each subpart contains a pointer to the next lower node in the tree and the tree is traversed using these pointers. Note that while each part "knows" what its children are, it does not know what its parent is. This is necessary so that the manufacturing process that uses the same part multiple places in the tree can be represented. In a future version of the program, it would be useful to have all of these parts in an array as well as connected via pointers. That data structure would make tasks such as flushing the temporary demand for all parts or freeing all of the allocated memory more straightforward.

The list of part numbers and the list that keeps track of the current location in the tree are stored in a doubly linked list. Several functions that involve the creation, maintenance, navigation, and destruction of doubly linked lists are found in *dlinkedlist.c.* Prototypes for these functions are in *dlinkedlist.h.*

4.3 Program Control

The control of the program starts in *control.c.* The main function calls helper functions to perform the tasks. The first task performed is reading all of the data in from the input file. This is accomplished by the functions in *readdata.c* and *readinfo.c.* Then the function demand in *demand.c* is called to determine how much of each part to start producing in each period. After *demand.c* finishes, *writer.c* contains a function to output the results to the user and *cleanup.c* contains code to free all of the relevant memory.

4.4 Constructing the Data Tree

The function *readinfo* is called until the (–1) at the end of the input file is returned. Each time *readinfo* is called, it adds the information for another node in the tree. It first reads the number of the part to be added. If this part has been seen before, the function simply finds that part in the tree and sets the pointer in the appropriate part's subparts array to point to the existing part. Otherwise, the function seeks the appropriate spot in the tree where it needs to add in the information and calls *readdata* to actually allocate space for the new part and read the information from the file. If a part number is given which has no proper location in the tree, *readinfo* exits with a status of –1 and the program terminates. When the –1 at the end of the file is read, *readinfo* exits with a status of 1 and the control passes back to *control.c.*

Two helper functions are key in the operation of *readinfo.c.* The first is *seeker.* This function takes as input the part which is the parent (final product) and any element of a doubly linked list which contains the location of the part to be returned (actually a pointer to that part is returned). Each element in the doubly linked list contains the number of the branch to be taken at each juncture in the tree starting on the left with branch number 1. The first element in this list will be 1 for the parent. The next element would be 1 to specify the first subpart on the left, then 2 for the next subpart from the left and so on.

The other key helper function is *seek_partnum.* It takes as input the parent part and the part number of any part in the tree. It then returns a pointer to the first part in the tree which has that part number. The function *seek_partnum* starts with parent and checks to see if it has the part number it is looking for and if not, calls the function *next_part.* The *next_part* function takes as input the parent part and any element of the doubly linked list which points to the part we are currently at. It then returns the next part in the tree as follows. First it checks to see if there are any subparts to the current part. If there are subparts, it returns the first subpart and exits. If there are no subparts, it returns to the part above the current one in the tree (it does this using *seeker* and the location doubly linked list). Then it checks to see if the part it was at was the last subpart of the current part. If it is not, the function returns the next subpart. If the part it was at was the last subpart of the current part, it moves to the next higher part in the tree and repeats the above process. The only exception is if it determines that it needs to move to a higher part, but finds itself at the parent of the tree. In this case an error message is printed and execution ceases.

4.5 Determining the Production

After all of the necessary data is read into the tree, the *control* function moves to find the optimal time to order the necessary amounts of all of the parts. The function responsible for performing this task is in *demand.c.* It takes as input the parent part of the tree, the number of periods, an integer array that is the number of periods long that upon output will contain the amount of the final product produced in each week, and the doubly linked list containing all of the part numbers.

The *demand* function works as follows. First it checks to see if the current demand for the parent part is less than the current inventory of that part. If so, the function simply decreases the amount of that part's inventory in all future weeks, increments the amount of the final part output in that week and moves to the next week. If the current demand for the final product is greater than the current inventory of that part, the program moves back the number of periods specified by the parent part's lead time and increments the array needed in the appropriate week for each subpart by the number of that part needed to meet the order (the amount of the part we need to produce multiplied by the amount of the subpart necessary to create one of the final product) It also increments the array *temp_start_prod* for the current part in the same week by the number of parts that need to be produced. If in attempting to do this, the program goes back before the first time period, it realizes that the order cannot be met, so it calls the function flush which sets all of the *temp_start_production* arrays to zero and all of the subparts' demand arrays to zero. After this call to flush, it moves one lot's worth of final product demand from the current week to the next week and attempts this process again. The program traverses through the tree making sure that all of the demanded parts are available for each week. When this happens, it calls *commit* which adds each entry in each part's *temp_start_prod* to the corresponding entry in the *starting_production* array, sets all entries of every subpart's needed array to zero and adjusts the inventories accordingly. This process is repeated for each week until the last week.

4.6 Writing the Output and Cleaning Up

After the *demand* function has determined the amount of each part that needs to be produced in each week, the *control* function calls *writer* that loops over all of the data and prints out a summary of the orders for the user. Then the function in *cleanup.c* is called to free all of the memory used in the program. This is non trivial because if the tree is deleted in the wrong order, all references to a part may be removed before the part can be accessed and its memory freed. The program does this dereferencing by making sure that it is always deleting the bottom left part in the tree.

5. Future Improvements

5.1 Input File Improvements

5.1.1 Icon-Based Bill of Materials

A second-generation input program would be best designed with an icon-based Bill of Materials that allowed the user to draw the BOM and input necessary data for each part. This would eliminate the most tedious, mistake-prone part of the input file. Also, a user would be able to identify mistakes more easily than with the current file. This icon-based system would more easily identify part/subpart relationships. This is especially valuable the more complicated the Bill of Materials becomes.

5.1.2 Batching Rule

The current program allows for only one type of batching rule (FOP). A future draft should incorporate other ordering rules as mentioned in Section 2.2. The user would designate the batching rule by entering a number at the beginning of the program that will indicate which rule to implement. The ordering rules will be described (and numbered) in the input manual

5.2 Output File Improvements

Some users might find it more convenient if the output file were converted into an easy-to-use spreadsheet. This could aid in the information manipulation necessary for MRP Nervousness study.

6. Conclusion

The program outlined above fulfills the initial requirements of the MRP Nervousness project. The program requires the user to create an input file that defines the Bill of Materials and other support information required for MRP calculations. The MRP program will perform the necessary calculations and produce an output file that clearly displays all relevant information including production, inventory levels, and ordering times for each part. There are multiple improvements that can be made to produce a more user-friendly yet more complex program. This program will be an important aid in the study of MRP Nervousness.

7. Bibliography

Greenwood, D.R. (2001). Sources and Propagation of Schedule Volatility in an MRP System. Cambridge, MA: Massachusetts Institute of Technology Leaders for Manufacturing.

Hopp, W.J., & Spearman, M.L. (1996). Factory Physics: Foundations of Manufacturing Management. Boston, MA.: The McGraw-Hill Companies, Inc.

Appendix A

The appendix contains the system of programs that comprise the greater MRP program.

A.1 demand.c

This function does all of the work to determine how much of everything needs to be produced in a give week.

```
#include "inventory.h"
 #include <stdio.h>
 #include <stdlib.h>

 int demand(struct part *parent, int period, int *output, struct listelement
 *partnumbers) {
   int i, j, k, finished, to_make, on_hand, status;
   struct listelement *location;
   struct part *current;


   for (i = 0; i < period; i++) {
/*   for(i = 0; i < 4; i++) {   */
 weekbeginning:
     current = parent;
     location = createdlinkedlist();
     location->data = 1;
     finished = 0;
     while (!finished) { /* loop until we can meet demand for this period */
       /* if we can cover production from our inventory */
       if (*(parent->on_hand+i) >= *(parent->needed+i)) {
         status = destroydlinkedlist(location);
         *(output+i) = *(parent->needed+i);
         /* if not the last week, subtract amount demanded from this week and all
future week's inventory */
         if (i != period) {
           for (j = i; j < period; j++) {
             *(parent->on_hand+j) = *(parent->on_hand+j) - *(parent->needed+i);
           }
         }
         finished = 1;
         break;
       } else { /* we can't cover production from our inventory */
         /* if we needed to start production before week 1, we can't make
          * the order, so push one lot back to the next week and see if
          * we can make a reduced order. (flush does this)
          */
         if (i - parent->lag_time < 0) {
           flush(parent, i, partnumbers);
           goto weekbeginning;
         } else {
           /* put appropriate number in the right week's temp_start_prod,
            * increase demand in the proper weeks for the subparts and
            * move on to the subparts.
         */
        to_make = *(parent->needed+i) - *(parent->on_hand+i);
        /* make sure to_make corresponds to an integer number of lots */
        if (to_make % parent->lot_size != 0) {
          to_make = ((to_make / parent->lot_size) + 1) * parent->lot_size;
        }
        *(parent->temp_start_prod+i - parent->lag_time) = \
          *(parent->temp_start_prod+i - parent->lag_time) + to_make;
        for (j = 0; j < parent->num_subparts; j++) {
          *((parent->subparts+j)->subp->needed + i - parent->lag_time) = \
            *((parent->subparts+j)->subp->needed+i-parent->lag_time) +
\
            to_make * (parent->subparts+j)->quantity;
        }
      }
      /* now deal with the demand in the subparts */
      while((current = next_part(parent, location)) != NULL) {
        /* need to look at every week prior to this one
         * so we'll get all of the demand right
         */
        for (j = 0; j <= i; j++) { /* j is the week we're looking at right now
*/
          /* figure out how much we really have on hand */
/*         printf("current week = %d, week = %d, partnum = %d, ", i+1, j+1,
current->partnum); */
          on_hand = *(current->on_hand+i);
          for (k = 0; k < (j-current->lag_time); k++) {
            on_hand = on_hand + *(current->temp_start_prod+k);
          }
          for (k = 0; k < (j-1); k++) {
            on_hand = on_hand - *(current->needed+k);
          }
/*         printf("on hand = %d, needed = %d\n", on_hand, *(current->needed+j)); */
          /* see if how much we have on hand this week will cover our demand */
          if (on_hand >= *(current->needed+j)) {
            continue;
          } else { /* need to make some more */
            /* if we can't make demand for this subpart we'll have to call
             * flush again and decrease our final demand, starting the whole
             * process over for this week.
             */
            if (j - current->lag_time < 0) {
              flush(parent, i, partnumbers);
              goto weekbeginning;
            } else {
              /* put appropriate number in the right week's temp_start_prod,
               * increase demand in the proper weeks for the subparts and
               * then move on.
               */
              to_make = *(current->needed+j) - on_hand;
              /* make sure to_make corresponds to an integer number of lots */
              if (to_make % current->lot_size != 0) {
                to_make = ((to_make / current->lot_size) + 1) * current->lot_size;
              }
              *(current->temp_start_prod +j - current->lag_time) = \
                *(current->temp_start_prod+j - current->lag_time) +
to_make;
                for (k = 0; k < current->num_subparts; k++) {
                  *((current->subparts+k)->subp->needed+j- current->lag_time) = \
                    *((current->subparts+k)->subp->needed+j- current->lag_time) + \
                    to_make * (current->subparts+k)->quantity;
                }
              }
            }
          }
        }
      }
      finished = 1;
      commit(parent, location, i, period);
    }
    if(status != NULL) {
      status = destroydlinkedlist(location);
    }
    *(output+i) = *(parent->needed+i);

  }

  return 0;
}
```

A.2 makefile.c

This function is useful in making compilation easier.

```
CC=/usr/bin/gcc
 CFLAGS=-g -Wall -pedantic -ansi

 all: main dlinkedlist
 main: control readinfo readdata demand next_part seek_partnum seeker unique
 dlinkedlist writer commit flush cleanup
      $(CC) $(CFLAGS) -o main.exe control.o readinfo.o readdata.o demand.o \
      next_part.o seek_partnum.o seeker.o unique.o dlinkedlist.o flush.o commit.o \
      writer.o cleanup.o; clear
control: control.c inventory.h
      $(CC) $(CFLAGS) -c control.c
readinfo: readinfo.c inventory.h
      $(CC) $(CFLAGS) -c readinfo.c
readdata: readdata.c inventory.h
      $(CC) $(CFLAGS) -c readdata.c
demand: demand.c inventory.h
      $(CC) $(CFLAGS) -c demand.c
next_part: next_part.c inventory.h
      $(CC) $(CFLAGS) -c next_part.c
seek_partnum: seek_partnum.c inventory.h
      $(CC) $(CFLAGS) -c seek_partnum.c
seeker: seeker.c inventory.h
      $(CC) $(CFLAGS) -c seeker.c
unique: unique.c inventory.h
      $(CC) $(CFLAGS) -c unique.c
```

```
dlinkedlist: dlinkedlist.c dlinkedlist.h
	$(CC) $(CFLAGS) -c dlinkedlist.c
flush: flush.c inventory.h
	$(CC) $(CFLAGS) -c flush.c
commit: commit.c inventory.h
	$(CC) $(CFLAGS) -c commit.c
writer: writer.c inventory.h
	$(CC) $(CFLAGS) -c writer.c
cleanup: cleanup.c inventory.h
	$(CC) $(CFLAGS) -c cleanup.c
clean:
	rm -f main.exe *.o *~
```

A.3 control.c:

This is the main thread of the program. It calls all of the helper routines.

```
#include "inventory.h"
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
  FILE *input;
  struct part *parent;
  struct part *current;
  int period, i;
  int *temp;
  int *real_demand;
  struct listelement *location;
  struct listelement *partnumbers;

  /* open the file that was named on the command line to read data from */
  if((input = fopen(argv[1], "r")) == NULL) {
    fprintf(stderr, "You must give the name of the input file on the command
line.\n");
    exit(2);
  }

  /* read in from file the number of periods */
  fscanf(input, "%d", &period);
  temp = calloc(period, sizeof(int));
  real_demand = calloc(period, sizeof(int));

  /* read into a temp array the demand for the final item in all weeks */
  for (i = 0; i < period; i++) {
    fscanf(input, "%d", (temp+i));
    *(real_demand+i) = *(temp+i);
  }



  /* initialize the doubly linked lists for part numbers and location */
  location = createdlinkedlist();
```

```
    location->data = 0;
    partnumbers = createdlinkedlist();


    parent = (struct part *) malloc(sizeof(struct part));
    current = parent;
    /* read all info about parts from file */
    while(readinfo(parent, current, period, input, location, partnumbers) == 0);
    fclose(input);

    /* move data from temporary array into the proper data structure */
    for (i = 0; i < period; i++) {
      *(parent->needed+i) = *(temp+i);
    }
    free(temp);

    /* call function which calculates the demand for all periods from
     * the given info
     */

    temp = calloc(period, sizeof(int));
    demand(parent, period, temp, partnumbers);

    /* call function which writes the demands to stdout */
    writer(parent, partnumbers, temp, period, real_demand);

    /* call function which cleans up all temporary memory */
    cleanup(parent, location, partnumbers, period);
    free(temp);
    free(real_demand);

    return 0;
  }
```

A.4 dlinkedlist.h

This is the header file containing the prototypes for the functions in dlinkedlist.c.

```
#include <stdlib.h>
#include <stdio.h>

struct listelement {
  struct listelement *backptr;
  struct listelement *forwardptr;
  int data;
};

struct listelement
 *createdlinkedlist(void);

int destroydlinkedlist(struct listelement *);

struct listelement
```

```
  *lastelement(struct listelement *);

struct listelement
  *firstelement(struct listelement *);

struct listelement
  *nextelement(struct listelement *);

struct listelement
  *previouselement(struct listelement *);

struct listelement
  *lengthen(struct listelement *);

struct listelement
  *shorten(struct listelement *);

struct listelement
  *removeelement(int data, struct listelement *anyelement);
```

A.5 cleanup.c

This program frees all memory used in the program. This includes the tree structure and the doubly linked lists.

```
#include <stdio.h>
#include "inventory.h"

int cleanup(struct part *parent, struct listelement *location, struct
listelement *partnumbers, int period) {
  struct part *current;
  int i, bottom, done;

  /* free all memory associated with the tree */
  done = 0;
  while(!done) {
    location = lastelement(location);
    while(location->backptr != location) {
      location = shorten(location);
    } location->data = 1;

    current = parent;
    /* traverse down the left side of the tree until we hit the bottom */
    bottom = 0;
    while (!bottom) {
      if (current->num_subparts == 0) {
        bottom = 1;
      } else if (unique(current->subparts->partnum, partnumbers)) {
        for (i = 0; i < current->num_subparts; i++) {
          *(current->subparts + i) = *(current->subparts + i + 1);
        }
        current->num_subparts = current->num_subparts - 1;
      } else {
            current = current->subparts->subp;
            location = lengthen(location);
            location->data = 1;
          }
        }
        /* free all memory associated with this element */
        if (current->partnum != 1) {
          free(current->subparts);
          free(current->needed);
          free(current->on_hand);
          free(current->starting_production);
          free(current->temp_start_prod);
        }
        if (current == parent) { /* if this is the parent we're done */
          removeelement(current->partnum, partnumbers);
          free(current);
          done = 1;
          continue;
        }
        /* if it's not go to the element that was one higher than this */
        removeelement(current->partnum, partnumbers);
        free(current);
        location = shorten(location);
        current = seeker(parent, location);
        /* make it's subparts so that they are shifted toward the left
         * and decrement the number of subparts */
        for (i = 0; i < (current->num_subparts - 1); i++) {
          *(current->subparts + i) = *(current->subparts + i + 1);
        }
        current->num_subparts = current->num_subparts - 1;
      }

      /* free memory associated with the location doubly linked list */
      destroydlinkedlist(location);

      /* we're done */
      return 0;
    }
```

A.6 commit.c

This will be called to finalize all changes made during a given pass of the demand algorithm and will be called once per pass.

```
#include "inventory.h"
 #include <stdio.h>
 #include <stdlib.h>

 /* sets starting_production += temp_start_prod in all weeks
  *
  * need to change inventory (on_hand) cumulatively in all weeks, so for each
 entry
  * in temp_start_prod, need to increase the inventory in all weeks later than
```

```
(current week + lag_time)
 * by the ammount temp_start prod
 *
 * need to treat inventory (on_hand) in same way when we're subtracting the
 amount demanded for
 * production.  So on_hand in all weeks from the current week to the end will be
 decrimented
 * by the value in demand for the current week.
 *
 * sets temp_start_prod = 0 for all elements in all weeks up to current week
 * sets demand = 0 for all elements except the parent for all weeks.
 */

int commit(struct part *parent, struct listelement *location, int week, int
period) {
  int i, j;
  struct part *current;

  while(location->backptr != location) {
    location = shorten(location);
  }
  location->data = 1;

  for(i = 0; i < week; i++) {
    /* deal with changing starting_production */
    *(parent->starting_production + i) = *(parent->starting_production + i) \
                                 + *(parent->temp_start_prod + i);
    /* deal with changing on_hand to account for temp_start_prod */

/*   printf("\nweek = %d\n", i+1);
    for (j = 0; j < period; j++) {
      printf("week = %d, on hand = %d\n", j+1, *(parent->on_hand+j));
    }*/

    if (i+parent->lag_time < period) {
      for (j = i; (j + parent->lag_time) < period; j++) {
        *(parent->on_hand + j + parent->lag_time) = *(parent->on_hand + j +
parent->lag_time) \
                                          +
*(parent->temp_start_prod + i);
      }
    }

/*     for (j = 0; j < period; j++) {
      printf("week = %d, on hand = %d\n", j+1, *(parent->on_hand+j));
    } */

    /* set temp_start_prod = 0 in all weeks */
    *(parent->temp_start_prod+i) = 0;
  }

  /* deal with changing on_hand to account for demand */
  for(i = week; i < period; i++) {
      *(parent->on_hand + i) = *(parent->on_hand + i) - *(parent->needed + week);
  }
  if (*(parent->on_hand+i) < 0) {
    fprintf(stderr, "Commit was called, but in week %d, the inventory of part
#%d %s", \
                              i+1, parent->partnum, "would become
negative\n");
    exit(2);
  }

  current = parent;
  while((current = next_part(parent, location)) != NULL) {
    for (i = 0; i <= week; i++) {
      /* deal with changing starting_production by temp_start_prod in all weeks */
      *(current->starting_production + i) = *(current->starting_production + i) \
                               + *(current->temp_start_prod + i);
      /* deal with changing on_hand for temp_start_prod */
      if (i + current->lag_time < period) {
        for (j = i; (j+current->lag_time) < period; j++) {
          *(current->on_hand+j + current->lag_time) = *(current->on_hand+j +
current->lag_time) \
                                        +
*(current->temp_start_prod + i);
        }
      }
      /* deal with changing on_hand to account for demand */
      if (i < period) {
        for (j = i; j < period; j++) {
          *(current->on_hand+j) = *(current->on_hand+j) - *(current->needed+i);
        }
        /*
        if (*(current->on_hand+i) < 0) {
          fprintf(stderr, "Commit was called, but in week %d, the inventory of
part #%d %s", \
                        i+1, current->partnum, "would become
negative\n");
          exit(2);
        }
         */
      }
      /* set temp_start_prod = 0 in all weeks */
      *(current->temp_start_prod+i) = 0;
      /* set needed = 0 in all weeks */
      *(current->needed+i) = 0;
    }
  }

  while(location->backptr != location) {
    location = shorten(location);
  }
  location->data = 1;

  return 0;
}
```

A.7 dlinkedlist.c

This contains several functions to make working with double linked lists possible.

```
#include "dlinkedlist.h"

/* create the first element of a doubly linked list */
struct listelement
  *createdlinkedlist(void) {
    struct listelement dummy;
    struct listelement *temp;

    temp = calloc(1,sizeof(dummy));
    if (temp == NULL) return NULL;
    temp->backptr = temp;
    temp->forwardptr = NULL;

    return temp;
  }

/* take any element of a doubly linked list and destroy the whole thing */
int destroydlinkedlist(struct listelement *anyelement) {

   anyelement = lastelement(anyelement);

   while(anyelement != NULL) {
     anyelement = shorten(anyelement);
   }
   return 0;
}

/* return the last element of the doubly linked list */
struct listelement
  *lastelement(struct listelement *anyelement) {

    if (anyelement->forwardptr == NULL) {
      return anyelement;
    } else {
      return lastelement(anyelement->forwardptr);
    }

    return NULL; /* if we get here, we're in trouble */
  }

/* return the first element of the doubly linked list */
struct listelement
  *firstelement(struct listelement *anyelement) {

    if (anyelement->backptr == anyelement) {
      return anyelement;
    } else {
      return firstelement(anyelement->backptr);
    }
    return NULL; /* if we get here, we're in trouble */
  }

/* go forward one element */
struct listelement
  *nextelement(struct listelement *currentelement) {

    if (currentelement->forwardptr == NULL) {
      return NULL;
    } else {
      return currentelement->forwardptr;
    }
  }

/* go back one element */
struct listelement
  *previouselement(struct listelement *currentelement) {

    if (currentelement->backptr == currentelement) {
      return NULL;
    } else {
      return currentelement->backptr;
    }
  }

/* add one element to end of doubly linked list */
struct listelement
  *lengthen(struct listelement *anyelement) {
    struct listelement *dummy;
    struct listelement dumber;
    struct listelement *temp;

    dummy = lastelement(anyelement);

    temp = calloc(1, sizeof(dumber));
    if (temp == NULL) return NULL;
    temp->backptr = dummy;
    temp->forwardptr = NULL;
    dummy->forwardptr = temp;
    temp->data = -1;

    return temp;
  }

/* remove one element from end of doubly linked list */
struct listelement
  *shorten(struct listelement *anyelement) {
    struct listelement *dummy;
    struct listelement *returnme;

    dummy = lastelement(anyelement);

    returnme = dummy->backptr;

    if (dummy == returnme) {
      free(dummy);
      return NULL;
    } else {
      free(dummy);
      returnme->forwardptr = NULL;
      return returnme;
    }
  }

/* remove an element containing (data) from doubly linked list */
struct listelement
  *removeelement(int data, struct listelement *anyelement) {
    struct listelement *temp;

    anyelement = lastelement(anyelement);
    while(anyelement->data != data) {
      if (anyelement->backptr == anyelement) {
        anyelement = NULL;
        fprintf(stderr, "The dlinked list did not have an element containing %d,
cowardly dying.", \
                data);
        exit(2);
      } else {
        anyelement = previouselement(anyelement);
      }
    }
    if(anyelement->backptr == anyelement) { /* we're at top */
      if (anyelement->forwardptr == NULL) { /* we're also at bottom */
        destroydlinkedlist(anyelement);
        return(NULL);
      } else { /* we're at top, but not bottom */
        anyelement = anyelement->forwardptr;
        free(anyelement->backptr);
        anyelement->backptr = anyelement;
        return(anyelement);
      }
    } else if (anyelement->forwardptr == NULL) { /* we're at bottom */
      anyelement = shorten(anyelement);
      return(anyelement);
    } else {  /* we're somewhere in the middle */
      temp = anyelement;
      anyelement->backptr->forwardptr = anyelement->forwardptr;
      anyelement->forwardptr->backptr = anyelement->backptr;
      anyelement = anyelement->forwardptr;
      free(temp);
      return(anyelement);
    }
    return NULL;
}
```

A.8 inventory.h

This is a header file that contains a few struct definitions and prototypes for all of the main functions of the program.

```
#include <stdio.h>
 #include <stdlib.h>
 #include "dlinkedlist.h"

 struct subpart {
   int partnum;        /* the number of the subpart */
   struct part *subp;  /* a pointer to the subpart */
   int quantity;  /* the quantity of this subpart required to make the assembly
 */
 };

 struct part {
    int partnum;               /* unique identifier for each part */
    int lot_size;              /* number of parts we produce per run */
    int lag_time;              /* amount of time it takes between starting
 production and getting the parts finished */
    int num_subparts;          /* number of subparts required to make this
 item */
    struct subpart *subparts;     /* Contains the part itself if this part
 requires no assembly */
    int *on_hand;              /* inventory of the part in a particular period
 */
    int *needed;               /* how many of this part are demanded in a
 particular period */
    int *starting_production;     /* how many of this part we're starting
 production for this period */
    int *temp_start_prod;        /* useful because may miss order this period
 and have to recalculate */
 };

 int readinfo(struct part *, struct part *, int, FILE *, struct listelement *,
 struct listelement *);
 int unique(int partno, struct listelement *partnumbers);
 struct part *seeker(struct part *parent, struct listelement *location);
 int readdata(struct part *current, int length, int partnumber, FILE *input);
 struct part *seek_partnum(struct part *parent, int partnum);
 struct part *next_part(struct part *, struct listelement *);
 int demand(struct part *parent, int, int *, struct listelement *);
 int flush(struct part *parent, int, struct listelement *);
 int commit(struct part *parent, struct listelement *location, int, int);
 int writer(struct part *parent, struct listelement *, int *, int, int *);
 int cleanup(struct part *parent, struct listelement *, struct listelement *,
 int);
```

A.9 flush.c

This is called whenever demand for a week can't be met and some demand needs to be moved to the next week. It also clears all temporary arrays in the product tree.

```
#include "inventory.h"
 #include <stdio.h>
 #include <stdlib.h>
 /* sets parent->*(demand+week) -= lot_size if it is greater than lot size
  * and to 0 if else.  It also sets parent->*(needed+week+1) += lot_size if the
  * previous week lost lot size and += whatever it lost otherwise.  It also
  * goes through and sets temp_start_production = 0 for all elements in all
  * weeks, as well as setting needed = 0 for all elements except the parent
  * for all weeks.
  */

 int flush(struct part *parent, int week, struct listelement *partnumbers) {
   int i, temp;
   struct part *current;
   struct listelement *temp_partnumbers;

   temp_partnumbers = partnumbers;

   current = parent;
   if (*(current->needed+week) >= current->lot_size) {
     *(current->needed+week) = *(current->needed+week) - current->lot_size;
     *(current->needed+week+1) = *(current->needed+week+1) + current->lot_size;
   } else {
     temp = *(current->needed+week);
     *(current->needed+week) = 0;
     *(current->needed+week+1) = *(current->needed+week+1) + temp;
   }
   for (i = 0; i <= week; i++) {
     *(current->temp_start_prod+i) = 0;
   }

   while (partnumbers->forwardptr != NULL) {
     partnumbers = partnumbers->forwardptr;
 /*     printf("partnumber = %d\n", partnumbers->data); */
     current = seek_partnum(parent, partnumbers->data);
 /*     for (i = 0; i <= week; i++) {
         printf("week = %d, needed = %d, temp_start_prod = %d\n", i+1,
 *(current->needed+i), *(current->temp_start_prod+i));
     } */
     for (i = 0; i <= week; i++) {
       *(current->temp_start_prod+i) = 0;
       *(current->needed+i) = 0;
     }
   }

   partnumbers = temp_partnumbers;

   return 0;
 }
```

A.10 readinfo.c

This is the function that determines where to put a new part into the tree and has logic to deal with repeated parts etc. It also calls readdata.c to actually read the information into the tree.

```
#include "inventory.h"
#include <stdio.h>
#include <stdlib.h>

int readinfo(struct part *parent, struct part *current, int length, FILE *input,
\
                struct listelement *location, struct listelement *partnumbers) {
  int i;
  int temp;
  int correct, newpart;

  /* have we come to the end of the file, if so exit */
  fscanf(input, "%d", &temp);
  if (temp == -1) {
    return 1;
  }

  /* this is the first part, act accordingly */
  if (location->data == 0) {
    location->data = 1;
    partnumbers->data = temp;
    readdata(current, length, temp, input);

  } else { /* this is not the first part we've come across */
    newpart = unique(temp, partnumbers); /* 1 if part is unique, 0 else */
    correct = 0;

    /* search to find where this part goes in the tree. Compare the
     * part number in temp with the subpart numbers in the part pointed
     * to by *location. If the partnumber is not already in our list of
     * *partnumbers, create a new location in the tree, extend *location
     * and *partnumbers appropriately and then read the data from the file
     * to fill out the new data. If the partnumber is not unique, don't
     * extend partnumber or location, and simply make the pointer in
     * the current part which points the subpart pointer to the right location
     */
    while (!correct) {
            current = seeker(parent, location);
            for (i = 0; i < current->num_subparts; i++) {
              if (temp == (current->subparts+i)->partnum && \
          (current->subparts+i)->subp == NULL) {
                correct = 1;
                if (newpart) { /* then we'll need to read in the data */
                        (current->subparts+i)->subp = (struct part *) malloc(sizeof(struct part));
                        partnumbers = lengthen(partnumbers);
                        partnumbers->data = temp;
                        location = lengthen(location);
                        location->data = i+1;
```

```
                              current = (current->subparts+i)->subp;
                              readdata(current, length, temp, input);
                              break;
                        } else { /* just update the pointer for the location of this subpart */
                              (current->subparts+i)->subp = seek_partnum(parent, temp);
                              break;
                        }
                    }
                  }
                  if (correct != 1) {
                    location = lastelement(location);
                    if (location->backptr != location) {
                      location = shorten(location);
              } else {
                      correct = -1;
                    }
          }
      }
      if (correct == -1) {
                  fprintf(stderr,"There was an error in the input file.\n \
                      Could not find part number %d (readinfo)\n", temp);
                  exit(2);
      }
  }
  return 0;
}
```

A.11 readdata.c

This function reads the data for a given part into the tree.

```
#include "inventory.h"
#include <stdio.h>
#include <stdlib.h>

/* allocates memory for the arrays in the part passed to it, and fills
 * them with the given partnumber, and what it reads from the file for:
 * lot_size, lag_time, num_subparts, on_hand[0] and the partnumbers
 * and quantity for each subpart.
 */

int readdata(struct part *current, int length, int partnumber, FILE *input) {
  int temp, i;

  current->on_hand = calloc(length, sizeof(int));
  current->needed = calloc(length, sizeof(int));
  current->starting_production = calloc(length, sizeof(int));
  current->temp_start_prod = calloc(length, sizeof(int));
  current->partnum = partnumber;
  fscanf(input, "%d", &current->lot_size);
  fscanf(input, "%d", &current->lag_time);
  fscanf(input, "%d", &temp);
  current->num_subparts = temp;
  fscanf(input, "%d", current->on_hand);
  for (i = 0; i < length; i++) {
    *(current->on_hand+i) = *current->on_hand;
  }

  current->subparts = calloc(temp, sizeof(struct subpart));
  for (i = 0; i < temp; i++) {
    fscanf(input, "%d %d", &(current->subparts+i)->partnum, \
         &(current->subparts+i)->quantity);
    (current->subparts+i)->subp = NULL;
  }

  return 0;
}
```

A.12 writer.c

This function prints the results of the simulation to the screen.

```
#include "inventory.h"
#include <stdio.h>

int writer(struct part *parent, struct listelement *partnumbers, int *made, int
period, int *real_demand) {
  struct part *current;
  struct listelement *temp_partnumbers;
  int i, any;

  printf("Below is a list of how many of the final part were output each
week\n");
  for(i = 0; i < period; i++) {
    printf("Week %d    output = %d, modified demand = %d, demanded = %d\n",
i+1, *(made+i), *(parent->needed+i), *(real_demand+i));
  }

  printf("Below is a listing of how many of each part we need to \n start
producing each week.\n");

  temp_partnumbers = firstelement(partnumbers);

  while(temp_partnumbers != NULL) {
    current = seek_partnum(parent, temp_partnumbers->data);
    any = 0;
    for (i = 0; i < period; i++) {
             if (*(current->starting_production+i) != 0) {
               if (any == 0) {
                 printf("Part Number %d\n", temp_partnumbers->data);
                 any = 1;
               }
               printf("Week %d:  %d\n", i+1, *(current->starting_production+i));
             }
    }
    temp_partnumbers = nextelement(temp_partnumbers);
  }

  printf("Below is a listing of the inventory of each part in each week.\n");
  temp_partnumbers = firstelement(partnumbers);
  while(temp_partnumbers != NULL) {
    current = seek_partnum(parent, temp_partnumbers->data);
    printf("Part Number %d, Inventory:\n", temp_partnumbers->data);
    for (i = 0; i < period; i++) {
            printf("Week %d:   %d\n", i+1, *(current->on_hand+i));
    }
    temp_partnumbers = nextelement(temp_partnumbers);
  }

  return 0;
}
```

A.13 seek_partnum.c

Given a number of a part in the tree, this function returns a pointer to the first part in the tree that matches that part number.

```
#include "inventory.h"
#include <stdio.h>
#include <stdlib.h>

/* runs through the tree and finds the part with the number specified
 * in the argument.  Returns NULL if it can't find that part number.
 */

struct part *seek_partnum(struct part *parent, int partnum) {
  struct listelement *location;
  struct part *current;
  int correct;

  location = createdlinkedlist();
  location->data = 1;

  correct = 0;
  current = seeker(parent, location);
  while (!correct) {
    if (current->partnum == partnum) { /* if we've found it, exit */
            destroydlinkedlist(location);
            return current;
    } else { /* go to the next part in the tree */
            current = next_part(parent, location);
            if (current == NULL) {
              fprintf(stderr, "The part number %d was not in the tree! \
                  (seek_partnum) \n", partnum);
              destroydlinkedlist(location);
              exit(2);
            }
    }
  }
  return 0;
}
```

A.14 next_part.c

This function returns the next part in the tree (see the programmer's manual for more information).

```
#include "inventory.h"
#include <stdio.h>

/* takes as input the parent and the dlinkedlist pointing to the part in the
 * tree where we are right now,
 * and returns the next part in the tree as well as updating the dlinked
 * list to point to that next part.  The next part is either
 * the first subpart of the input part, the next subpart of the input
 * part's parent, or the program goes back up the tree until if finds
 * a part whose subparts haven't been looked at and returns the next
 * subpart of that part.  (think I could say part one more time in this
 * part of the comment :-P )
 */

struct part *next_part(struct part *parent, struct listelement *location) {
  struct part *next_part;
  int finished, temp;

  location = lastelement(location);
  next_part = seeker(parent, location);
  location = lastelement(location);

  if (next_part->num_subparts != 0) { /* the first subpart is untouched so
return the first subpart */
    location = lengthen(location);
    location->data = 1;
    next_part = next_part->subparts->subp;
    return next_part;
  }

  /* otherwise this is the bottom of a branch */
  finished = 0;
  while(!finished) { /* we'll go upwards until we reach a part with some
untouched subparts */
    location = lastelement(location);
    temp = location->data;
    if (location->backptr == location) {  /* we've at the parent and can't go
higher */
            finished = -1;
    } else { /* go up to the part above and see if it has any more unused
subparts */
            location = shorten(location);
            next_part = seeker(parent, location);
            if (temp < next_part->num_subparts) {
              location = lengthen(location);
              location->data = temp + 1;
              if ((next_part->subparts + temp)->subp == NULL) {
                fprintf(stderr, "I've come to an incomplete part of the tree\n \
                        While looking for the next part, so I'm dying.
(next_part)\n");
                    exit(2);
                  } else {
                    next_part = (next_part->subparts + temp)->subp;
                    finished = 1;
                  }
                }
      }
  }
  if (finished == -1) {
    return NULL;
  } else {
    return next_part;
  }
}
```

A.15 seeker.c

Given a location in the tree, this function returns a pointer to the part associated with that location.

```
/* returns a pointer to the element in the tree specified by location */
#include "inventory.h"
#include <stdio.h>

struct part *seeker(struct part *parent, struct listelement *location) {
  struct part *temp;

  temp = parent;
  location = firstelement(location);

  while(location->forwardptr != NULL) {
    location = nextelement(location);
    if (((temp->subparts+location->data-1)->subp == NULL) && \
        (temp->subparts+location->data-1)->subp->num_subparts != 0) {
            fprintf(stderr, "I've been asked to seek to a part of the tree that\n \
                isn't there, shamelessly dying. (seeker)\n");
            exit(2);
    } else {
            temp = (temp->subparts+location->data-1)->subp;
    }
  }

  return temp;
}
```

A.16 unique.c

This function determines whether a given part is already listed in the tree.

```
/* returns 0 if the part number is contained in the list, 1 if it is not */
#include "inventory.h"
#include <stdio.h>

int unique(int partno, struct listelement *partnumbers) {
  struct listelement *temp;

  temp = partnumbers;
  temp = firstelement(partnumbers);

  while(temp->forwardptr != 0) {
    if (temp->data == partno) {
      return 0;
    } else {
      temp = nextelement(temp);
    }
  }

  if (temp->data == partno) {
    return 0;
  } else {
    return 1;
  }
}
```