CS2303 In Class Exercises

Jan 24, 2020

January 11, 2020

Abstract

Today's goal is that you improve at design, the most creative part of the overall process of solving programming problems. In particular, become with familiar with graphs, trees and binary trees.

Name and WPI email: Kush Shah kshah2@wpi.edu

1 Design

One design principle is to have form follow function.
There are many commonalities in the function a programming solution might have.
One way to accomplish design is to know about a repertoire of solutions to problems, and to select among them, modify them and combine them to come up with a new design.

2 Components of Design

We design data structures, and we design algorithms that work with data structures.
When we design data structures, we have algorithms that will work with them in mind.

3 Designing Data Structures

Data structures perform the job of receiving information, and of being able to subsequently produce that information.
Features of giving and returning information include how much space and how much time it takes, to perform these functions.
The features can be expect to have different degrees of relevance for any particular application.
Therefore, there are multiple data structures that have been developed, and

characterized for the amount of time it takes to enter and retrieve information, and the amount of space needed.

Example:
Consider an item's age. We can enter an age (in units of days), update it every night.
Then, whenever that age is retrieved, we can read it.
The memory is small, the work is daily.
By contrast, we can enter the birthday, and when the age is desired, we can calculate it on retrieval.
There is a tradeoff between calculation and memory space.

One part of designing an application is to understand the relative value of resources like computation steps (which are transformed to time), and memory, and to have knowledge of the multiple data structures, and choose among, and possibly modify, or make entirely new datastructures, so as to use sparingly the resource that is relatively costly.

These resource utilization attributes of datastructures are studied in the context of the size of the problem being solved.

4 Example Data Structures

4.1 Lots of Individual Variables and Arrays

Variables in the programming language are a means of using memory. When we have multiple items of the same datatype, it is notationally convenient to have one name, the array's name, and an index to go with the array name, to identify which the multiple items we are storing or reading. The compiler maintains a table for matching variable names with memory addresses, called the symbol table. For a variable that is not an array, the name can be used to look up the address. For a variable name that refers to an array, the variable name is mapped to the address of the start of the collection of variables, and the index is added to that starting address, after having been scaled to the number of bytes for the datatype of the elements of the array.

The time utilization of looking up an element in an array is thus the time to multiply the index by the scale factor, and add the initial address. It is not a function of the number of elements in the array, so long as that arithmetic fits into reasonable number of bytes. Therefore we say, the time utilization of array reading, and writing, is of the order of a constant, denoted O(1).

The memory utilization of storing an array depends in a linear fashion upon the number of elements in the array. This is denoted O(n).

There are data structures that can take advantage of extensive redundancy in the data elements that would otherwise be stored in an array. There are also data structures that can take advantage of data mostly zero (sparse arrays), or some other constant. There are data structures that can take advantage of absence of as much as half of the data most of the time (dynamic arrays).

4.2 Linked Lists

When we do not know how big a collection of data might become, and/or we like the versatility of being able to add or remove data from the interior points in a collection, without having to move lots of others to maintain an array without gaps, there is an alternative called linked list.

We will study the linked list idea, and practice it, so that it becomes a comfortable choice of data structure.

Linked lists may be singly or doubly linked.

4.2.1 Singly Linked Lists

The linked list can be divided into two parts, namely the "infrastructure" part, that provides the linkages, and the "element" part, that stores the items to be retrieved.

An example of the infrastructure part:

```
struct LLNode;
typedef struct
{
    struct LLNode* next;
    Payload* payP;
}LLNode;
```

We have worked with user-defined types; we have seen the typedef struct keywords before.
The struct in the example above introduces a new feature, the ability to have self-referential data structures.
What makes this data structure self-referential is that the datatype of a component refers to the datatype being defined.
Looking at the end of the datastructure definition, we see that this datastructure is called LLNode.
Looking inside the datastructure definition, we see that the phrase LLNode* appears inside.
If we were to omit the line "struct LLNode;" then the first time the compiler encountered LLNode would be inside the structure definition, so it would be an unknown term, which is disadvantageous for compilation. For this reason, we include the line "struct LLNode".
The implication of this line is that the compiler can expect to become informed of the defintion of this struct, later. This line acts like an I-owe-you (IOU) promissory note for the definition of struct LLNode.

Once this IOU has been issued, it will be necessary to incorporate the keyword struct when the datatype name is used.

The infrastructure part refers to the data element it carries, as a payload. This can be thought of as a general term, and set to whatever datatype is being stored in the list list.

Example:

```
typedef struct
{
    int roomNumber;
    float treasure;
    bool searched;
}Room;

typedef Room Payload;

struct LLNode;
typedef struct
{
    struct LLNode* next;
    Payload* payP;
}LLNode;
```

Note in the example above that Room is defined. The statement "typedef Room Payload;" tells the compiler that Payload is what Room is. (Think define 'something-you-know' 'something new'.)

Exercise: (As always, if you have questions, ask. After this occasion, this material becomes background knowledge.)

1. Extend the datatype for Room to include whether or not the room has an alarm, and whether or not the alarm was set off.

```
typedef struct
{
int roomNumber;
float treasure;
bool searched;
bool alarm;
bool alarmSound;
}Room;
```

2. Does the extension of the Room datatype cause any change in the infrastructure part?

no chenge other then the fact that there are 2 more fields under room and consequently under payload.

4.2.2 Traversing the Singly Linked List

The definition of a list is:
list := <empty list > || list-element, list

This is a recursive definition. It means that a list may be empty, or a list may have an element in it (when the second mention of list is the empty list), or the list may be longer.

This definition provides a useful method of designing functions that work with lists, an idea to which we shall return later.

Both the empty list and the list with one element contain one instance of the LLNode. The empty list is distinguished from the list with one element by having the Payload pointer field, payP, be zero for the empty list.

Suppose we are given a location of a list element. It need not be the first element in the list. To access the payload being carried by that node, we use the "payload" field. To move to the next element, we update the location we were given to that found in the "next" field.

We can tell we are at the tail end of the list because for that element, the "next" field is zero.

Example:

```
LLNode* temp = lp;
while(temp->next)
{
    temp=(LLNode*)temp->next;
}
//now temp points to the last element
```

Note that in the above code, a variable, temp, able to hold the address of (point to) a linked list node is initialized to a variable lp. The next field will be zero if and only if we are at the last element. The statement inside the block of the while statement updates the temp variable to contain the address of the next element in the list. If more processing is desired at a node before moving to the next node, that processing can be put into the block of the while statement.

Exercise: (As always, if you have questions, ask. After this occasion, this material becomes background knowledge.)

1. Extend the code for traversing the linked list, so that, at each node, the room number is printed, for example, with printf.

```
while(temp->next){
temp=(LLNode*)temp->next;
printf("Room: %d",room);
}

LLNode* temp = lp;
while(temp->next){
i
temp=(LLNode*)temp->next;
} else {
temp=(LLNode*)temp->prev;
}
}
```

2. Extend the code for traversing the linked list, so that, at each node, the room number is printed, only if it is greater than the previous room number.

```
while(temp->next){
if(room>lastRoom{
printf("Room %d",room);
}
lastRoom = room;
temp=(LLNode*)temp->next;
}
```

Suppose we want to traverse in the other direction. We can add a field to make this possible.

4.2.3 Doubly Linked List

The doubly linked list adds a field to the infrastructure datatype.

```
struct LLNode;
typedef struct
{
    struct LLNode* next;
    struct LLNode* prev;
    Payload* payP;
}LLNode;
```

Note in the above that one field has been added to the singly linked list datatype, that it is also a struct LLNode pointer, and that it is named prev. The names LLNode, next, prev, Payload and payP are arbitrary, and may be changed without affecting functionality.

With this structure, traversal may move forward as before, but also backward. Just as the tail end of the list is distinguished by having the next field be zero, the head end of the list is distinguished by the prev field be zero.

Exercise: (As always, if you have questions, ask. After this occasion, this material becomes background knowledge.)

1. Extend the code for traversing the linked list, so that, after reaching the tail end of the list, the direction of traversal is reversed, and the start of the list is reached by moving through each element. Count the elements in the list.

```
LLNode* temp = lp;
if(next!=0){
while(temp->next){
temp=(LLNode*)temp->next;
}
}else {
while(temp->prev){
temp=(LLNode*)temp->prev;
}
}}
```

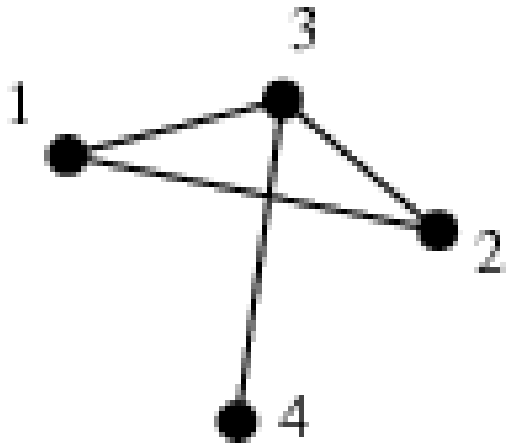


Figure 1: This graph is connected, because any vertex can be reached from any other vertex by following edges.

4.3 Graphs

Graphs are a marvelous abstraction, but we shall only discuss some items relevant for datastructures and algorithms. Graphs are a useful data structure for algorithms. Many problems can be abstracted to graphs. A graph has two parts: The first part is a set of nodes, we shall imagine them to have unique identifiers, such as the numbers 0 through n-1, when there are n nodes (also called vertices). The second part is a set of edges. Each edge consists of a pair of node identifiers. If the pair is ordered, the edge is also an arc (It has a direction, given by the ordering of the pair.)

Example:

In the example above there are four vertices and four edges. The vertices are 1, 2, 3 and 4. The edges are (1,2), (1,3), (2,3) and (3,4; all are undirected.

One use of a graph is to show the direct (wired) interconnections between computers. The computers are the nodes, and the communication link wires are the edges. A molecule can be represented by a graph. The vertices are the atoms and the edges are the bonds. Subway maps are graphs. Floorplans are graphs: the rooms are vertices and the hallways are edges.

Because graphs are so useful in computation, there are datastructures for them. One such datastructure is the adjacency matrix. To visualize an adjacency matrix, imagine a two dimensional array. Each dimension has one value for each node. The elements within the array are either zero or one. If the nodes are directly connected (have an edge) the element is one. Otherwise it is zero.

Exercise: (As always, if you have questions, ask. After this occasion, this

material becomes background knowledge.)

1. If every vertex is considered to be connected to itself, is the main diagonal necessarily all ones?

   yes

2. If a graph has no directed edges, is the matrix necessarily symmetrical about the main diagonal?

   yes

We can use code to allocate and initialize an adjacency matrix. Example:

```
typedef struct
{
   int n;
   int* edgesP;
}AdjMat;

void init(AdjMat* adjMP)
{
    int ncols = adjMP->n;
    for(int row = 0; row<ncols; row++)
    {
        for(int col = 0; col<ncols; col++)
        {
            *((adjMP->edgesP)+(row*ncols)+col)= 0;
        }
    }
}
```

Note in the example above the number of vertices is provided in the AdjMat data structure. The formula for addressing each cell in the matrix uses address arithmetic. The array is stored one row after another, and the row is treated as a

list of column cells. The variable adjMP− >edgesP is the address of where the array starts. The formula " *((adjMP− >edgesP)+(row*ncols)+col)" starts where the array starts, and moves through each row, one column at a time, through the whole array.

We can use code to set an edge between two vertices. Example:

```
void setEdge(AdjMat* adjMP, int row, int col)
{
    int ncols = adjMP->n;
    int* arrayBeginning = adjMP->edgesP;
    *(arrayBeginning + (ncols*row) +col) = 1;
    *(arrayBeginning + (ncols*col) + row) = 1;
}
```

Note that this code is assuming the edges are undirected.

Exercise: (As always, if you have questions, ask. After this occasion, this material becomes background knowledge.)

1. What is it about this code that makes the edge undirected?

   the matrix is symetric so that means that both (row, col) and (col, row) have a 1

2. What change this code would make the edge directed?

   removeing the last line of the function would make it directed

4.4 Trees

Graphs that satisfy additional properties are trees. A tree can be constructed as follows:
Start with a single node. That is a tree. Add one node (the result is not a tree) then add an edge between the nodes. That is a tree. Each time another node is added, it should be connected to the tree by a single edge. It will remain a tree. Note that a tree is a graph that contains no cycles. That is, for any two nodes, there is only one path between them. If there were more than one

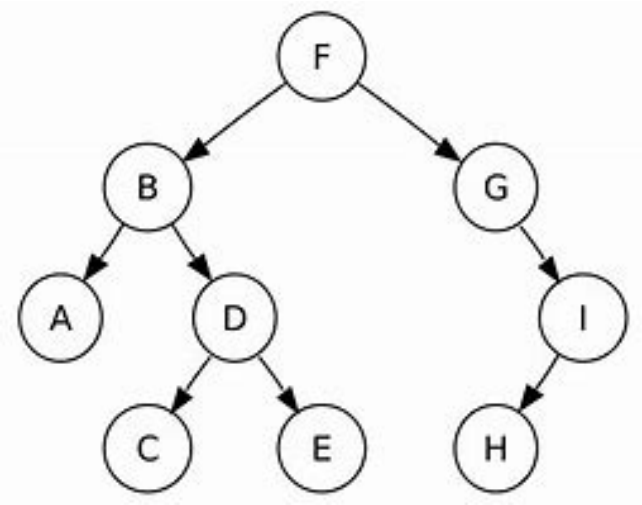


Figure 2: A binary tree satisfies the property that each node has at most two child nodes.

path, we could start at the first node, follow one path to the second node, and a different path back to the first node, and the two paths taken together would be a cycle.

Trees are useful datastructures. There are algorithms that exploit trees, and these algorithms can be recursive. Note that starting with a node, we can find each of its child nodes (reachable in one edge, and not the the node's parent). Each child node is a subtree. This makes the datastructure recursive.

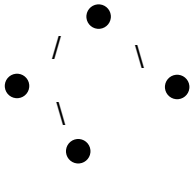
Trees can be extracted from graphs. A tree that has been extracted from a graph has all the same node. If the graph has cycles, just enough edges are removed so that there are no cycles.

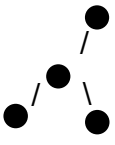
4.4.1 Binary Trees

Suppose a tree's nodes each had at most two children. Then it would be a binary tree. The two child nodes are called left and right. Example:

Exercise: (As always, if you have questions, ask. After this occasion, this material becomes background knowledge.)

1. Draw two different binary trees.





The data structure for a binary tree would need to provide for both children. Example:

```
struct LLNode;
typedef struct
{
    struct LLNode* left;
    struct LLNode* right;
    Payload* payP;
}LLNode;
```

Note that the example data structure is not really different from that used for a doubly linked list. Only the field names have changed. If it is desirable to traverse the tree both downwards from the root to the leaves, and also upwards from the leaves toward the root, a field would be added.

```
struct LLNode;
typedef struct
{
   struct LLNode* parentP;
   struct LLNode* left;
   struct LLNode* right;
   Payload* payP;
}LLNode;
```

Exercise: (As always, if you have questions, ask. After this occasion, this material becomes background knowledge.)

1. How would you set the values in the datastructure to indicate that a node is a leaf node?

   left and right would be 0