Edinburgh Napier University

SET08122 Algorithms & Data Structures

Lab 5 - Data Structures #3

Dr Simon Wells

1 Aims

Our goal this week is to become familiar with a range of basic data structures in our chosen language. If you are already familiar with the structures explored in the activities section below for your chosen language then it is worth either trying a different language, so that you have something to compare against, or else, exploring the standard library for your chosen language to discover something you didn't already know. Finally, any given introduction to data structures will only and can only scratch the surface; so if you know all the structures from this lab, find a data structure that is totally new to you to investigate.

At the end of the practical portion of this topic you will be able to:

- Implement a simple associative data structure
- Implement a simple binary tree
- Implement a simple directed graph using an adjanceny list implementation

Unfortunately we are starting to get slightly below the surface of data structures and these waters can be quite deep. For each of today's data structures there are multiple ways that each structure can be conceptualised, multiple ways that it can be implemented, and usually, multiple standard algorithms for handling the resulting structure. This is why we will return to each of these structures later in the module to revist them, with a particular focus on searching, sorting, traversing, and calculating results.

2 Structured Activities

We're going to implement a simple associative data structure, a hash-table that will enable us to store key-value pairs and look up values by supplying a key. We'll then implement a simple binary tree, before finally implementing a simple directed graph. As each of these is quite a challenging, but powerful, structure, you should do additional background research to round out your understanding. In each case we will revist the topic to explore additional features later in the module.

2.1 Associative Data Structures

These are ways to collect data that are linear but not sequential. Instead of relying upon contiguous placement in memory, or one element pointing to the next in sequence, an associative structure relies upon an identifier, a *key* to specify the location of the requested element, and it's associated *value* in the collection. So associative structures are key-value collections. The idea is that when a key is supplied, the data associated with the key is returned. We can consider array access by index to be a simple version of this, supply the array index, your key, and you can retrieve whatever is stored in the array at that point. However the innovation here is to consider if the key is something other than just an array index, what about an ID? or a postcode? or a telephone number? or an email address? Lot's of ways that we might want to organise our data in the real world.

You might already have seen similar data structures in other languages for example, the dictionary in Python. The use of key value stores is so pervasive that there are even entire databases built around the idea.

Let's build a simple associative data structure. Our API will include the following core functions:

1. insert
2. delete
3. search

You should have started to notice, across all of the data structures that we've developed so far, that there are some commonalities to our APIs. We nearly always need a method for each of getting data into out structure, out of our structure, and for searching for our data once it is within the structure.

We often also nee need some ancillary functions such as, in this case, a display() function to traverse our structure and print the contents to the screen. This isn't always essential, but is really useful, particularly whilst learning. We'll also implement an additional utility function called hashCode() that we'll use to help decide where to insert a new element into our array. This is the first time we've had to create a utility or helper function for a data structure that isn't part of our core API. Interestingly, hashCode() is a function that we'll revisit later in the module because there are many different strategies for implementing it, but no single best, or perfect, way to do so.

Let's start with the display() function so that it's ready when we need it:

```
1  void display(void)
2  {
3      int i;
4      for(i=0; i<SIZE; i++)
5      {
6          if(hashArray[i] != NULL)
7              printf(" (%d,%d)", hashArray[i] -> key, hashArray[i] -> data);
8          else
9              printf(" ~,~ ");
10     }
11     printf("\n");
12 }
```

Display just iterates through our structure, and for each element, we print out our key followed by our value as a comma-separate pair. This isn't the only way we could implement this, but it's enough for us to rapidly see what's stored within out structure.

Now we should really do our insert() function, but that will use the hashCode() function so let's address that first. The hashCode implementation is very simple, it merely returns a potential index value, or place to store the value, based upon the key *modulo* the size of our underlying storage (an array in this case).

```
1  int hashCode(int key)
2  {
3      return key % SIZE;
4  }
```

There are a lot of hashing algorithms, many of which have various performance characteristics and we'll return to the topic later in the module. In a perfect world though, we'd also have a perfect hashing algorithm, a function that for any key would map that key to only one slot in our underlying storage, avoiding collisions with other data that also maps to the same slot. Similarly, we'd also want our dataset to only need underlying storage of exactly the same size as itself and that there'd be no clashes. Unfortunately there are no perfect hashing algorithms, but there are lots of ways to get better performance than we've achieved here. A key aspect of this is that the hash function should ideally give constant, or near constant, time access to any given value, regardless of how large the structure is or how much data is stored within it.

Now we have the preliminaries in place so that we can insert some data into our structure using an insert() function:

```
1  void insert(int key, int data)
2  {
3      struct DataItem *item = (struct DataItem*) malloc(sizeof(struct DataItem));
4      item -> data = data;
5      item -> key = key;
6
7      int hashIndex = hashCode(key);
8      while(hashArray[hashIndex] != NULL && hashArray[hashIndex] -> key != -1)
9      {
10         ++hashIndex;
11         hashIndex %= SIZE;
12     }
13     hashArray[hashIndex] = item;
14 }
```

To add a new member to our structure we first need to allocate space for it, then we need to set the key and value for our newly allocated data item struct to the values passed into insert() as arguments. Once we have our new data item we can then find a place to put it in the associative structure. We merely get the array index that is equal to the supplied key modulo the array size. If that space is taken then we merely increment the location that we are looking at until we find an empty slot.

Actually this implementation has a glaring ommission. What if our underlying array is full? What happens to the program if there is no place for an additional data item? How would you change the implementation to handle this situation? Perhaps you might want to keep track of how many slots you've *probed* and when you've probed a number of slots equal to the size of the underlying array then you know it is full and can return. Alternatively, perhaps you might want to keep track of each addition or removal in some form of tally that gives an idea of how full or empty the structure is?. Perhaps you might want to indicate to the calling function that it has been unsuccessful in inserting a new data item?

Once data is in our store we'll want to be able to retrieve it, so we need a search function:

```
1  struct DataItem* search(int key)
2  {
3      int hashIndex = hashCode(key);
4      while(hashArray[hashIndex] != NULL)
5      {
6          if(hashArray[hashIndex] -> key == key)
7              return hashArray[hashIndex];
8
9          ++hashIndex;
10         hashIndex %= SIZE;
11     }
12
13     return NULL;
14 }
```

Notice that this is a far from ideal way to retrieve our data, becuase there may be *collisions* amongst our keys, we might need to search several slots before we identify our key. We'll return to search later in the module and, together with a look at hash functions, we'll see if we can come up with some more performant methods for inserting and retrieving elements in an associative structure that will give constant time insertion and retrieval.

Finally, we need a delete() function so that we can remove old elements from the structure:

```
1  struct DataItem* delete(struct DataItem* item)
2  {
3      int key = item -> key;
4      int hashIndex = hashCode(key);
5
6      while(hashArray[hashIndex] != NULL)
7      {
8          if(hashArray[hashIndex] -> key == key)
9          {
10             struct DataItem *temp = hashArray[hashIndex];
11             hashArray[hashIndex] = NULL;
12             return temp;
13         }
14         ++hashIndex;
15         hashIndex %= SIZE;
16     }
17     return NULL;
18 }
```

Here we are essentially finding our element, the one we want to remove, then assigning the slot where it is found to NULL.

We can use our structure with some code in a main function to drive it.

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

struct DataItem* delete(struct DataItem*);
void display(void);
void insert(int, int);
int hashCode(int);
struct DataItem* search(int);

#define SIZE 10
struct DataItem
{
    int data;
    int key;
};

struct DataItem* hashArray[SIZE];
struct DataItem* tempItem;
struct DataItem* item;

int main(void)
{
    tempItem = (struct DataItem*) malloc(sizeof(struct DataItem));
    tempItem -> data  =-1;
    tempItem -> key = -1;

    display();

    insert(1, 20);
    insert(2, 70);
    insert(42, 80);
    insert(4, 25);
    insert(12, 44);
    insert(14, 32);
    insert(17, 11);
    insert(37, 97);
    insert(10, 0);
    display();

    item = search(37);
    if(item != NULL)
    {
        printf("Element found: %d\n", item->data);
    }
    else
    {
        printf("Element not found\n");
    }

    item = delete(item);
    if(item != NULL)
    {
        printf("Element found: %d\n", item->data);
    }
    else
    {
        printf("Element not found\n");
    }
    display();

    item = search(37);
    if(item != NULL)
    {
        printf("Element found: %d\n", item->data);
    }
    else
    {
        printf("Element not found\n");
    }
    display();

    return(0);
}
```

Play around with this. Again there is a full listing in Appendix A to help if you get confused assembling the associated data structure. You should notice that you can insert data, in this case an int representing a key and another int representing the value, into our underlying data structure, which is implemented using an array. As with our other data structures, such as the stack, queue, and deque, we can get different performance characteristics and a larger storage area, without having to resize, by using linked lists to implement our associative data structure but we'll return to this later.

2.2 Tree Data Structures

Think of a child's drawing of a tree. A single trunk that branches over and over again until we reach the leaves[1]. This is very similar to what we mean when we refer to a tree data structure, a single trunk, usually referrerd to as the *root*. The root can split into a number of brances, which can each in turn split into futher branches, over and over again, until we reach our leaves. The root, and each branching point, and each leaf are *nodes*. Nodes have two basic responsibilities, to hold a value and to point to child nodes (in the leaf direction. Think of the root direction as pointing towards the parents of any given node).

Just as the real world has many varieties of tree, each of which is subtly different from other trees, so the tree data structure can have many forms, as well as many different ways to implement each form. We are going to concentrate today on the binary tree. Something that is binary is composed of two things. In the context of a binary tree, the word binary refers to the branching factor of the tree. Each node can have no more than two branches, a branch to a left-child, and a branch to a right child. There are various forms of the binary tree which we could also implement, for example, trees that always try to balance the left and right children and to ensure that no single branch becomes longer than any other. However we'll just build a basic binary tree for now, one in which each node can hold some data and point to no more than two children.

For this we need a struct to hold the information associated with each node in our tree. This requires pointers from itself to a left-child and a right-child, as well as data that the node will store.Our data is thus stored within a tree-shaped branching organisation of data that is potentially spread across memory.

```
1 struct binary_tree_node
2 {
3     struct binary_tree_node *left_child;
4     struct binary_tree_node *right_child;
5     int data;
6 };
```

Notice that each of our child pointers just points to a structure which is of exactly the same type as the one that we are defining. This means that child nodes are no differnt to parent nodes, or the root node for that matter. They mainly differ based upon their relations to each other, but not due to the basic structure.

Before we do anything further we want to implement an insert function to give us a way to add new nodes to our tree.

```
1  void insert(struct binary_tree_node **node, int num)
2  {
3      if(*node == NULL)
4      {
5          *node = (struct binary_tree_node *) malloc (sizeof(struct binary_tree_node)
               );
6          (*node) -> left_child = NULL;
7          (*node) -> right_child = NULL;
8          (*node) -> data = num;
9      }
10     else
11     {
12         if (num < (*node) -> data)
13             insert( &((*node) -> left_child), num);
14         else
```

[1]This is obviously not during Winter (unless you were thinking of an evergreen).

```
15              insert(&((*node) -> right_child), num);
16      }
17  }
```

All we have done here is check whether our tree is empty, by checking whether the passed in node is pointing to NULL or not. If it is then we need to create a root node. Otherwise we need to traverse our tree until we get to a node that has an empty child slot, then we can insert our new data. However, notice that this is a recursive function, insert is calling insert again from within itself. So whilst we start with a call to insert, passing in the root node, we then proceed by passing in a child of our current node until we reach an empty slot. A second thing to note is that we are also selecting branches of our tree to traverse based upon the value of the data stored in each node. If our input data, the value to store, is less than our current value then we take the branch associated with the left child, otherwise we use the other branch. This means that smaller values will sort to one side of a node and larger values will sort to the other side. Note that this sorting is node-based and not tree based. Values don't sort to a side of the tree, but to a side of a parent node.

Once we have some data in our tree we need to be able to output the contents of our tree. We could call this function print, but we are not merely iterating over the contents of the tree like we would a list, we are moving through the tree, from node to node. We are *traversing* the tree, and printing out our data at each step.

```
1  void traverse(struct binary_tree_node *node)
2  {
3      if (node != NULL)
4      {
5          traverse(node -> left_child);
6          printf("%d\t", node -> data);
7          traverse(node -> right_child);
8      }
9  }
```

Of note is the fact that we have chosen an order in which to traverse the tree and that the traverse function is recursive, traverse calls itself, passing the new traverse call one of it's children each time. So for each call to traverse, there will be two further calls to traverse, one for each child. Note the ordder in which traverse is called, for any given node, we call traverse for the left child, then when that returns we print the value of the current node, then we call traverse for the right child. This means that the traverse function will travel all the way to the bottom of the left most part of the tree before even printing out it's first data value, before iteratively traversing the right hand child.

Once you have a working tree program, return to this function and investigate how different ordering of the iterative calls to traverse and print will lead to different orders of output of the contents of the tree. It's also worth adding in some extra print statements to help you track each stage of the traversal so that you can really understand what's happening here.

If we have data stored in a tree then we will probably want to find individual items of data within that collection. To achieve this we need to search, hence we need a search function.

```
1  void search(struct binary_tree_node **root, int num, struct binary_tree_node **
       parent, struct binary_tree_node **found_node, int *found_status)
2  {
3      struct binary_tree_node *temp;
4      temp = *root;
5      *found_status = FALSE;
6      *parent = NULL;
7
8      while(temp != NULL)
9      {
10         if(temp -> data == num)
11         {
12             *found_status = TRUE;
13             *found_node = temp;
14             return;
15         }
```

```
16          *parent = temp;
17          if(temp -> data > num)
18              temp = temp -> left_child;
19          else
20              temp = temp -> right_child;
21      }
22  }
```

Notice the similarities with traversal, we need to step through our tree, and we are using a simlar pattern to before, left child, then right child. At each node in the tree we are checking whether our search value is the same as the stored value.

Deleting a node is actually fairly complicated as we need to account for several situations, when the node has no children, when the node has just one child to the left, when the node has one child to the right, and when the node has two children. You should be able to see a pattern to how this is achieved in the following function. Notice that there are also checks for when the tree is empty, so there is nothing to delete, and when the item to delete isn't in the tree, in which case there is nothing to delete again.

```
1   void delete(struct binary_tree_node **root, int num)
2   {
3       int found;
4       struct binary_tree_node *parent, *search_node, *next;
5
6       if(*root == NULL)
7       {
8           printf("Tree is empty\n");
9           return;
10      }
11
12      parent = search_node = NULL;
13      search(root, num, &parent, &search_node, &found);
14
15      if(found == FALSE)
16      {
17          printf("Data not found\n");
18          return;
19      }
20
21      if(search_node -> left_child != NULL && search_node -> right_child != NULL)
22      {
23          parent = search_node;
24          next = search_node -> right_child;
25          while(next -> left_child != NULL)
26          {
27              parent = next;
28              next = next -> left_child;
29          }
30          search_node -> data = next -> data;
31          search_node = next;
32      }
33
34      if(search_node -> left_child == NULL && search_node-> right_child == NULL)
35      {
36          if(parent -> right_child == search_node)
37              parent -> right_child = NULL;
38          else
39              parent -> left_child = NULL;
40
41          free(search_node);
42          return;
43      }
44
45      if(search_node -> left_child == NULL && search_node -> right_child != NULL)
46      {
47          if (parent -> left_child == search_node)
48              parent -> left_child = search_node -> right_child;
49          else
50              parent -> right_child = search_node -> right_child;
51
52          free(search_node);
53          return;
54      }
```

```
55
56      if(search_node -> left_child != NULL && search_node -> right_child == NULL)
57      {
58          if(parent -> left_child == search_node)
59              parent -> left_child = search_node -> left_child;
60          else
61              parent -> right_child = search_node -> left_child;
62
63          free(search_node);
64          return;
65      }
66  }
```

Notice that when we delete a node, if the node has children then we have to reattach those children to the parent of the removed node. If there is just one child then this is straigthforward, but when there are two then this becomes slightly more complex and we must *rotate* the sub-tree (comprising the child and it's children so that one node is the new root of the subtree and can be attached to the parent. If we don't do this then we will have too many nodes to attach to the parent because a binary tree is restricted to two children per parent node.

Finally, a main function to test our tree implementation:

```
1   int main(void)
2   {
3
4       struct binary_tree_node *root;
5       int i=0;
6       int content[] = {11, 9, 13, 8, 10, 12, 14, 15, 7};
7       root = NULL;
8
9       while(i<=8)
10      {
11          insert(&root, content[i]);
12          i++;
13      }
14
15      traverse(root);
16      printf("\n");
17
18      delete(&root, 10);
19      traverse(root);
20      printf("\n");
21
22      delete(&root, 14);
23      traverse(root);
24      printf("\n");
25
26      delete(&root, 8);
27      traverse(root);
28      printf("\n");
29
30      delete(&root, 13);
31      traverse(root);
32      printf("\n");
33
34
35      return 0;
36  }
```

Notice that the implementation is reasonably straightforward, we just need a single root node as our entry point to our tree. We then add child nodes as required, respecting the two child limit of the binary tree, until we have added all of our data. For ease of demonstration I have just used an array of ints to feed the growth of our tree. Consider what you might have to do in order to add randomly generated int values into the tree instead.

2.3 Graph Data Structures

The graph is a very powerful and flexible structure for modelling real world problem domains, particularly relationships between members of a network. For example, modelling the members of a social network, or relationships between people in a family, or the roles of people in an organisation.

A graph is constructed from a set of vertices, sometimes referred to as node, and a set of edges that link the vertices together as required. Think of this as being like a linked list but the elements of the list can point to any number of other elements in the list.

For our graph implementation we are going to look at a directed graph. This is a graph in which the edges have a direction, e.g. vertice_1 points to vertice_2 but vertice_2 doesn't necessarily also point back to vertice_1. In an undirected graph, if there is an edge between vertice_1 and vertice_2 then the relationship is bi-directional rather than uni-directional.

Beacuse graphs can get very complicated, we are going to use a simpler method to implement our data structure. We are going to use *adjaceny lists*. This basically means that we have a number of simple linked lists, one for each vertex in our graph. Each vertex's list just stores references to the other vertices that it is pointing to. This means that for each vertex we can retrieve a list of it's neighbours, and if we have a list of all vertices then we can walk through our graph, visiting the vertices as required, for eaxmple, of we wanted to print out the graph.

Let's start with some structs to hold the various parts of our graph. We have a graph struct, a node struct, and an edge struct. The graph struct stores a list of all of the vertices that the graph contains. The node struct stores the id of a target vertex, in this case a simple int identifier, and a pointer to the first of it's neighbouring vertices, which in turn points to another neighbouring vertex, and so on until the list contains the complex set of neighbours for the original vertex. We do that for all of the vertexes so that each node struct essentially stores the neighbours of each individual vertex. Finally the edge struct stores the identified for a source and a destination vertex as simple ints. In a real world implementation we might have other data stored in each node besides just the *target* but for now we just want to capture the structure of a directed graph.

```
1  #define NUM_VERTICES 6
2
3  struct Graph
4  {
5      struct Node* head[NUM_VERTICES];
6  };
7
8  struct Node
9  {
10     int target;
11     struct Node *next;
12 };
13
14 struct Edge
15 {
16     int source;
17     int target;
18 };
```

Notice that we only really need the graph and node structs for our actual data structure. The edge struct is merely a convenience to help us to describe our graph in terms of a collection of relationships between nodes.

Given our structs, we now need a function to actually construct a graph. We want to pass in an array of edges which can then be used to construct our list of neighbour nodes. Each list is then added to our instantiation of the graph struct.

```
1  struct Graph* create_graph(struct Edge edges[], int num_edges)
2  {
3      int i;
4      struct Graph *graph = (struct Graph*) malloc(sizeof(struct Graph));
5
6      for(i=0; i< NUM_VERTICES; i++)
7      {
8          graph -> head[i] = NULL;
9      }
10
11     for(i=0; i< num_edges; i++)
12     {
13         int source = edges[i].source;
14         int target = edges[i].target;
```

```
15
16          struct Node* new_node = (struct Node*) malloc(sizeof(struct Node));
17          new_node -> target = target;
18          new_node -> next = graph -> head[source];
19          graph -> head[source] = new_node;
20      }
21
22      return graph;
23  }
```

A key thing to notice with the graph, which is different to everything that has come before, is that the graph is the first data structure that really doesn't have a naturally defined first element, or a root node. All nodes have the same status. Whilst we do have an ordering of our nodes as part of the implementation of the graph struct, this is merely because of the order of construction, and doesn't confer any special status upopn the node that happens to be vertex number one. As a result, the adjacency list implementation gives us direct access to each and every node. From any node, we can then traverse the graph by accessing the related nodes from our current location. In some graphs this could lead to a situation where there are some nodes that cannot be accessed from a given starting node.

Having constructed a graph, we probably want to print it out. Unfortunately printing a graph is difficult to do, and is usually best performed graphically. However, graph visualisation is a whole discipline of it's own. Instead, we'll just print out the adjacency lists for the vertices of our graph.

```
1  void print_graph(struct Graph* graph)
2  {
3    int i;
4    for (i = 0; i < NUM_VERTICES; i++)
5    {
6      struct Node* ptr = graph->head[i];
7      while (ptr != NULL)
8      {
9        printf("(%d -> %d)\t", i, ptr->target);
10       ptr = ptr->next;
11     }
12
13     printf("\n");
14   }
15 }
```

In this case, all we are doing is stepping through our array of vertices that are stored in the graph struct, and for each vertex, we are printing out it's adjacent nodes, those that it is directly connected to by edges.

Now we need a main() function to drive our program. We need to define a set of edges, then pass that to our graph creation function. Finally, we can print our graph to the screen in the form of an adjaceny list. Remember that you'll need to include the stdio.h and stdlib.h headers and also to include function prototypes for a complete program.

```
1  int main(void)
2  {
3      struct Edge edges[] =
4      {
5          { 0, 1 }, { 1, 2 }, { 2, 0 }, { 2, 1 }, { 3, 2 }, { 4, 5 }, { 5, 4 }
6      };
7
8      int n = sizeof(edges)/sizeof(edges[0]);
9
10     struct Graph *graph = create_graph(edges, n);
11
12     print_graph(graph);
13     return 0;
14 }
```

It is worth your while drawing out, on paper, how you think this graph will look. Remember, each number is the id of a vertex. and each $\rightarrow$ is a relationship pointing from a source to a target.

Just as weith associative structures and trees, we wil return to graphs later in the module, but with more of a focus on algorithms for traversing graphs and searching data within them. One of the things that graphs offer us, because of the complexity of potential relationships between individual nodes, are ways to extract various subsets of relationships. For example, the subset of relationships that give us all of the vertices that are connected to each other, or if we also store a weight associated with each relationship, the set of vertices and edges that give the minimum routes between a set of vertices.

As with other data structures, there are multiple competing methods for implementing graphs. Whilst we've chosen the adjacency list approach, adjaceny matrix approaches are also popular, utilising multi-dimensional array to represent the vertices and their relatinships. Some graph libraries, particulaly in other languages than C, take different approaches, for example, constructing large inter-related sets of objects, however this can be tricky to scale.

When dealing with real-world contexts that require a graph solution, the performance of the graph, in terms of how long it takes to compute something with the data stored within it can vary greatly depending upon both the scale of the data, how that data is organised, and how the organisation is translated into a graph representation.

As with all data structures that we've examined so far, there are many variations in terms of both abstract concepts and implementational approaches. Whether a given structure is appropriate to a given problem depends upon understanding the problem, then selecing an appropriate approach that can be justified. So don't just reach for a data structure because you know how to use it. Reach for a data structure because it will help you to efficiently, accurately, and robustly solve your problem.

3 Summary

The important takeaway here is that as we move further away from simple data structures that are mostly reliant on the organisation of physical memory, things get increasingly complicated. However, these complications mask great power and flexibilty.

With the associative data structure we are able to begin retrieving data from our structure, not based upon the value we are looking for, or its location in the structure, but based upon a key that has means in *association* with it's value.

With the tree we are starting to create a hierachical data structure in which, starting from a root node, we can branch off to a number of child nodes. Whilst the tree that we've implemented today is a binary tree, because it has two children, we can implement other trees to enable us to model our data in ways that match the natural structure of the data rather than the restrictions of the data structure itself.

With the graph we finally have an incredibly flexible data structure that is non-linear enabling any given node to *point* to any other node in an increasingly dense network of relationships. This gives us a really powerfull modelling technique for handling the natural structure of a large number of real world problems.

Part I
Appendices

A Associative Data Structure Source Code Listing

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 struct DataItem* delete(struct DataItem*);
5 void display(void);
```

```
6  void insert(int, int);
7  int hashCode(int);
8  struct DataItem* search(int);
9
10 #define SIZE 10
11 struct DataItem
12 {
13     int data;
14     int key;
15 };
16
17 struct DataItem* hashArray[SIZE];
18 struct DataItem* tempItem;
19 struct DataItem* item;
20
21 int main(void)
22 {
23     tempItem = (struct DataItem*) malloc(sizeof(struct DataItem));
24     tempItem -> data  =-1;
25     tempItem -> key = -1;
26
27     display();
28
29     insert(1, 20);
30     insert(2, 70);
31     insert(42, 80);
32     insert(4, 25);
33     insert(12, 44);
34     insert(14, 32);
35     insert(17, 11);
36     insert(37, 97);
37     insert(10, 0);
38     display();
39
40     item = search(37);
41     if(item != NULL)
42     {
43         printf("Element found: %d\n", item->data);
44     }
45     else
46     {
47         printf("Element not found\n");
48     }
49
50     item = delete(item);
51     if(item != NULL)
52     {
53         printf("Element found: %d\n", item->data);
54     }
55     else
56     {
57         printf("Element not found\n");
58     }
59     display();
60
61     item = search(37);
62     if(item != NULL)
63     {
64         printf("Element found: %d\n", item->data);
65     }
66     else
67     {
68         printf("Element not found\n");
69     }
70     display();
71
72     return(0);
73 }
74
75 struct DataItem* delete(struct DataItem* item)
76 {
77     int key = item -> key;
78     int hashIndex = hashCode(key);
79
80     while(hashArray[hashIndex] != NULL)
81     {
```

```
82          if(hashArray[hashIndex] -> key == key)
83          {
84              struct DataItem *temp = hashArray[hashIndex];
85              hashArray[hashIndex] = NULL;
86              return temp;
87          }
88          ++hashIndex;
89          hashIndex %= SIZE;
90      }
91      return NULL;
92  }
93
94  void display(void)
95  {
96      int i;
97      for(i=0; i<SIZE; i++)
98      {
99          if(hashArray[i] != NULL)
100             printf(" (%d,%d)", hashArray[i] -> key, hashArray[i] -> data);
101         else
102             printf(" ~,~ ");
103     }
104     printf("\n");
105 }
106
107 void insert(int key, int data)
108 {
109     struct DataItem *item = (struct DataItem*) malloc(sizeof(struct DataItem));
110     item -> data = data;
111     item -> key = key;
112
113     int hashIndex = hashCode(key);
114     while(hashArray[hashIndex] != NULL && hashArray[hashIndex] -> key != -1)
115     {
116         ++hashIndex;
117         hashIndex %= SIZE;
118     }
119     hashArray[hashIndex] = item;
120 }
121
122 int hashCode(int key)
123 {
124     return key % SIZE;
125 }
126
127 struct DataItem* search(int key)
128 {
129     int hashIndex = hashCode(key);
130     while(hashArray[hashIndex] != NULL)
131     {
132         if(hashArray[hashIndex] -> key == key)
133             return hashArray[hashIndex];
134
135         ++hashIndex;
136         hashIndex %= SIZE;
137     }
138
139     return NULL;
140 }
```

B Binary Tree Source Code Listing

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  #define TRUE    1
5  #define FALSE   0
6
7  struct binary_tree_node
8  {
9      struct binary_tree_node *left_child;
10     struct binary_tree_node *right_child;
11     int data;
```

```
};

void insert(struct binary_tree_node **, int);
void traverse(struct binary_tree_node *);
void search(struct binary_tree_node **, int, struct binary_tree_node **, struct
    binary_tree_node **, int *);
void delete(struct binary_tree_node **, int);

int main(void)
{

    struct binary_tree_node *root;
    int i=0;
    int content[] = {11, 9, 13, 8, 10, 12, 14, 15, 7};
    root = NULL;

    while(i<=8)
    {
        insert(&root, content[i]);
        i++;
    }

    traverse(root);
    printf("\n");

    delete(&root, 10);
    traverse(root);
    printf("\n");

    delete(&root, 14);
    traverse(root);
    printf("\n");

    delete(&root, 8);
    traverse(root);
    printf("\n");

    delete(&root, 13);
    traverse(root);
    printf("\n");


    return 0;
}

void insert(struct binary_tree_node **node, int num)
{
    if(*node == NULL)
    {
        *node = (struct binary_tree_node *) malloc (sizeof(struct binary_tree_node)
            );
        (*node) -> left_child = NULL;
        (*node) -> right_child = NULL;
        (*node) -> data = num;
    }
    else
    {
        if (num < (*node) -> data)
            insert( &((*node) -> left_child), num);
        else
            insert(&((*node) -> right_child), num);
    }
}

void traverse(struct binary_tree_node *node)
{
    if (node != NULL)
    {
        traverse(node -> left_child);
        printf("%d\t", node -> data);
        traverse(node -> right_child);
    }
}

void search(struct binary_tree_node **root, int num, struct binary_tree_node **
    parent, struct binary_tree_node **found_node, int *found_status)
```

```
{
    struct binary_tree_node *temp;
    temp = *root;
    *found_status = FALSE;
    *parent = NULL;

    while(temp != NULL)
    {
        if(temp -> data == num)
        {
            *found_status = TRUE;
            *found_node = temp;
            return;
        }
        *parent = temp;
        if(temp -> data > num)
            temp = temp -> left_child;
        else
            temp = temp -> right_child;
    }
}

void delete(struct binary_tree_node **root, int num)
{
    int found;
    struct binary_tree_node *parent, *search_node, *next;

    if(*root == NULL)
    {
        printf("Tree is empty\n");
        return;
    }

    parent = search_node = NULL;
    search(root, num, &parent, &search_node, &found);

    if(found == FALSE)
    {
        printf("Data not found\n");
        return;
    }

    if(search_node -> left_child != NULL && search_node -> right_child != NULL)
    {
        parent = search_node;
        next = search_node -> right_child;
        while(next -> left_child != NULL)
        {
            parent = next;
            next = next -> left_child;
        }
        search_node -> data = next -> data;
        search_node = next;
    }

    if(search_node -> left_child == NULL && search_node-> right_child == NULL)
    {
        if(parent -> right_child == search_node)
            parent -> right_child = NULL;
        else
            parent -> left_child = NULL;

        free(search_node);
        return;
    }

    if(search_node -> left_child == NULL && search_node -> right_child != NULL)
    {
        if (parent -> left_child == search_node)
            parent -> left_child = search_node -> right_child;
        else
            parent -> right_child = search_node -> right_child;

        free(search_node);
        return;
    }
```

```
161
162     if(search_node -> left_child != NULL && search_node -> right_child == NULL)
163     {
164         if(parent -> left_child == search_node)
165             parent -> left_child = search_node -> left_child;
166         else
167             parent -> right_child = search_node -> left_child;
168
169         free(search_node);
170         return;
171     }
172 }
```

C Directed Graph Source Code Listing

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  #define NUM_VERTICES 6
5
6  struct Graph
7  {
8      struct Node* head[NUM_VERTICES];
9  };
10
11 struct Node
12 {
13     int target;
14     struct Node *next;
15 };
16
17 struct Edge
18 {
19     int source;
20     int target;
21 };
22
23 struct Graph* create_graph(struct Edge[], int);
24 void print_graph(struct Graph*);
25
26 int main(void)
27 {
28     struct Edge edges[] =
29   {
30     { 0, 1 }, { 1, 2 }, { 2, 0 }, { 2, 1 },
31     { 3, 2 }, { 4, 5 }, { 5, 4 }
32   };
33
34   int n = sizeof(edges)/sizeof(edges[0]);
35
36   struct Graph *graph = create_graph(edges, n);
37
38   print_graph(graph);
39     return 0;
40 }
41
42 struct Graph* create_graph(struct Edge edges[], int num_edges)
43 {
44     int i;
45     struct Graph *graph = (struct Graph*) malloc(sizeof(struct Graph));
46
47     for(i=0; i< NUM_VERTICES; i++)
48     {
49         graph -> head[i] = NULL;
50     }
51
52     for(i=0; i< num_edges; i++)
53     {
54         int source = edges[i].source;
55         int target = edges[i].target;
56
```

```
57          struct Node* new_node = (struct Node*) malloc(sizeof(struct Node));
58          new_node -> target = target;
59          new_node -> next = graph -> head[source];
60          graph -> head[source] = new_node;
61      }
62
63      return graph;
64  }
65
66  void print_graph(struct Graph* graph)
67  {
68    int i;
69    for (i = 0; i < NUM_VERTICES; i++)
70    {
71      struct Node* ptr = graph->head[i];
72      while (ptr != NULL)
73      {
74        printf("(%d -> %d)\t", i, ptr->target);
75        ptr = ptr->next;
76      }
77
78      printf("\n");
79    }
80  }
```