Texas Tech University

A Course on Digital forensics

Memory Forensics

Module 2 – Data Structures

Akbar S. Namin, Spring 2018

CHAPTER 2 – DATA STRUCTURES

OUTLINE

CONTENT

- Data structures are the basic building blocks programmers use for implementing software and organizing how the program's data is stored within memory.
- It is extremely important for you to have a basic understanding of the common data structures.
- Leveraging this knowledge helps you to determine the most effective types of analysis techniques, to understand the associated limitations of those techniques, to recognize malicious data modifications.

1. Basic Data Types

- We build data structures using the basic data types that a particular programming language provides.
- We use the basic data types to specify how a particular set of bits is utilized within a program.

a. C Programming Language

- Given its usefulness for systems programming and its facilities for directly managing memory allocations, C is frequently encountered when analyzing the memory-resident state of modern operating systems.
- Table 2-1 shows basic data types for the C programming language.

Common Storage Sizes for C Basic Data Types

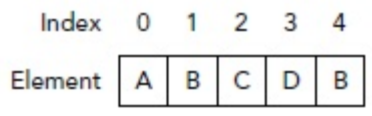
Type	32-Bit Storage Size (Bytes)	64-Bit Storage Size (Bytes)
`char`	1	1
`unsigned char`	1	1
`signed char`	1	1
`int`	4	4
`unsigned int`	4	4
`short`	2	2
`unsigned short`	2	2
`long`	4	Windows: 4, Linux/Mac: 8
`unsigned long`	4	Windows: 4, Linux/Mac: 8
`long long`	8	8
`unsigned long long`	8	8
`float`	4	4
`double`	8	8
`pointer`	4	8

b. Abstract Data Types

- Abstract data types provide models for both the data and the operations performed on the data.
- These models are independent of any particular programming language and are not concerned with the details of the particular data being stored.
- We will generically refer to the stored data as an *element*, which is used to represent an unspecified data type.
- This element could be either a basic data type or a composite data type.
- The values of some elements—pointers—may also be used to represent the connections between the elements.

c. Arrays

- The simplest mechanism for aggregating data is the *one-dimensional array*.
- This is a collection of <index, element> pairs, in which the elements are of a homogeneous data type.



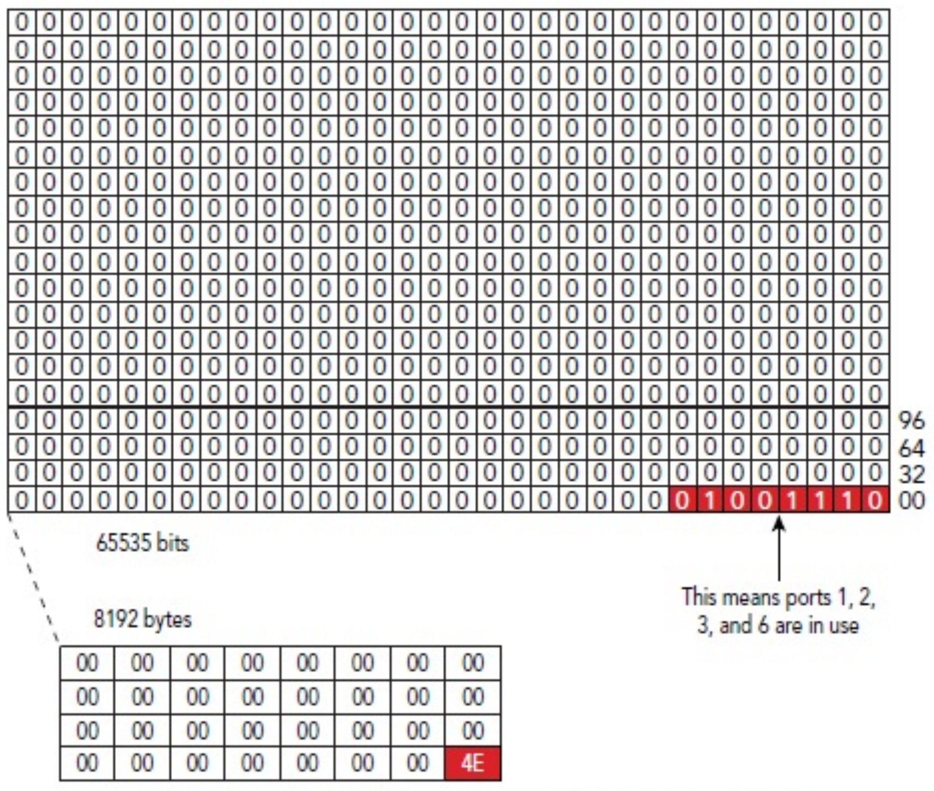
One-dimensional array example

- In Figure, you have an example of an array that can hold five elements.

- Arrays are frequently used because many programming languages offer implementations designed to be extremely efficient for accessing stored data.
- Because arrays are extremely efficient for accessing data, you encounter them frequently during memory analysis.

d. Bitmaps

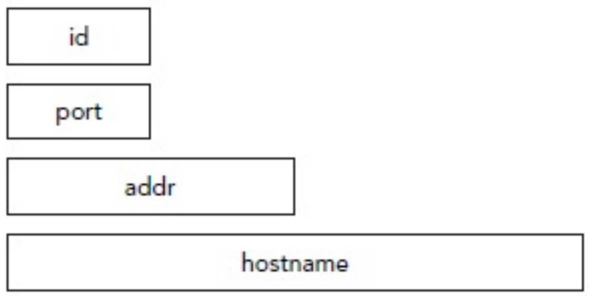
- An array variant used to represent sets is the *bitmap*, also known as the *bit vector* or *bit array*.
- In this instance, the index represents a fixed number of contiguous integers, and the elements store a boolean value {1,0}.
- They are stored as an array of bits, known as a *map*, and each bit represents whether one object is valid or not.
- Using bitmaps allows for representation of eight objects in one byte, which scales well to large data sets.



An example of a Windows bitmap of in-use network ports

e. Records

- Unlike an array that requires elements to consist of the same data type, a record can be made up of heterogeneous elements.
- It is composed of a collection of fields, where each field is specified by <name, element> pairs.
- Each field is also commonly referred to as a member of the record.



Network connection record example

- In the C programming language, records are implemented using structures.

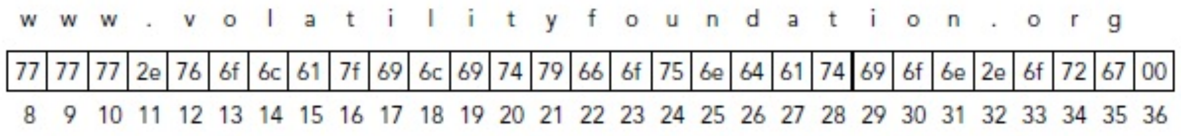
  struct Connection {
  short id;
  short port;
  unsigned long addr;
  char hostname[32];
  };

- The following is an example of how an instance of the data structure would appear in memory, as presented by a tool that reads raw data:

```
0000000: 0100 0050 ad3d de09 7777 772e 766f 6c61   ...P.>X.www.vola
0000010: 7469 6c69 7479 666f 756e 6461 7469 6f6e  tilityfoundation
0000020: 2e6f 7267 0000 0000                      .org....
```

f. Strings

- Stored elements are constrained to represent character codes taken from a predetermined character encoding. Similar to an array, a string is composed of a collection of <index, element> pairs.
- The first implementation we are going to consider is the C-style string. C-style strings where the elements are of type char—a C basic data type—and are encoded using the ASCII character encoding.
- For example, consider the data structure discussed previously for storing network information whose structure type was presented in Table 2-3. The fourth element of the data structure used to store the hostname is a C-style string.
- Figure shows how the string would be stored in memory using the ASCII character encoding.



w	w	w	.	v	o	l	a	t	i	l	i	t	y	f	o	u	n	d	a	t	i	o	n	.	o	r	g	
77	77	77	2e	76	6f	6c	61	7f	69	6c	69	74	79	66	6f	75	6e	64	61	74	69	6f	6e	2e	6f	72	67	00
8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36

Hostname represented in ASCII as a C-style string

- Strings are an extremely important component of memory analysis, because they are used to store textual data (passwords, process names, filenames, and so on).

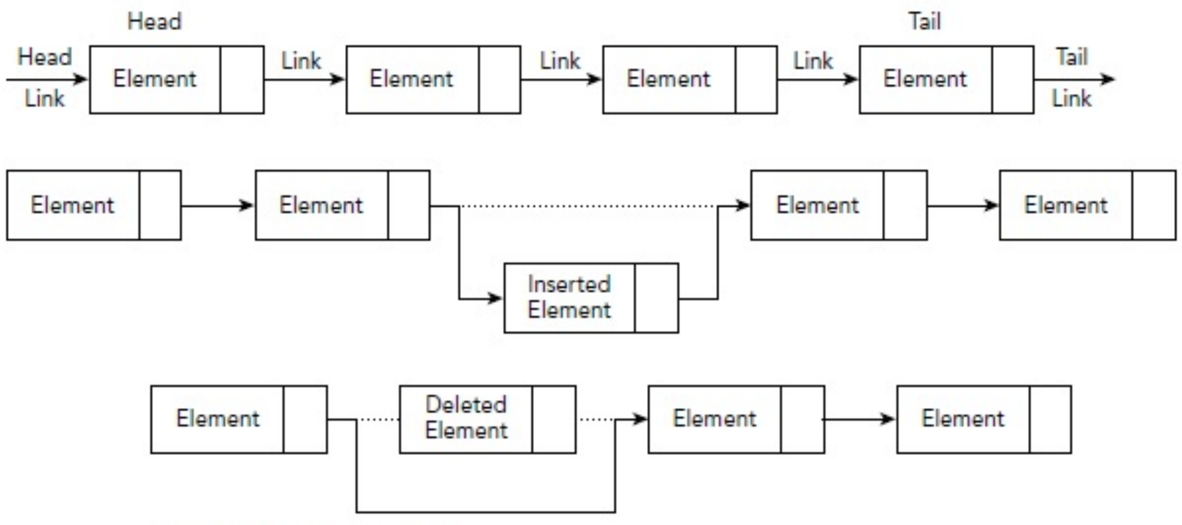
- While investigating an incident, you will commonly search for and extract relevant strings from memory. In these cases, the string elements can provide important clues as to what the data is and how it is being used.

g. Linked Lists

- A *linked-list* is an abstract data type commonly used for storing a collection of elements. Unlike fixed-size arrays and records, a linked-list is intended to provide a flexible structure. The structure can efficiently support dynamic updates and is unbounded with respect to the number of elements that it can store.

• Singly Linked List

- Each element of the singly linked list is connected by a single link to its neighbor and, as a result, the list can be traversed in only one direction.



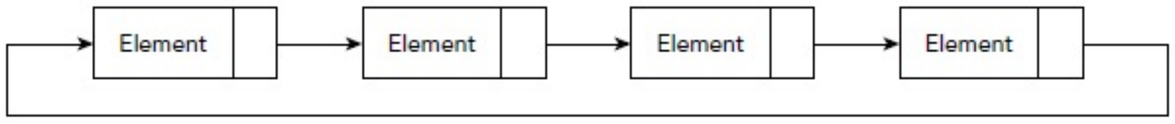
Singly-linked list example

• Doubly Linked List

- It is also possible to create a doubly linked list in which each element stores two pointers: one to its predecessor in the sequence and the other to its successor. Thus, you can traverse a doubly linked list both forward and backward.

• Circular Linked List

- It is called a circular linked list because the final link stored with the tail refers to the initial node in the list (list head). This is particularly useful for lists in which the ordering is not important.



Circular linked list example

• Embedded Doubly Linked List

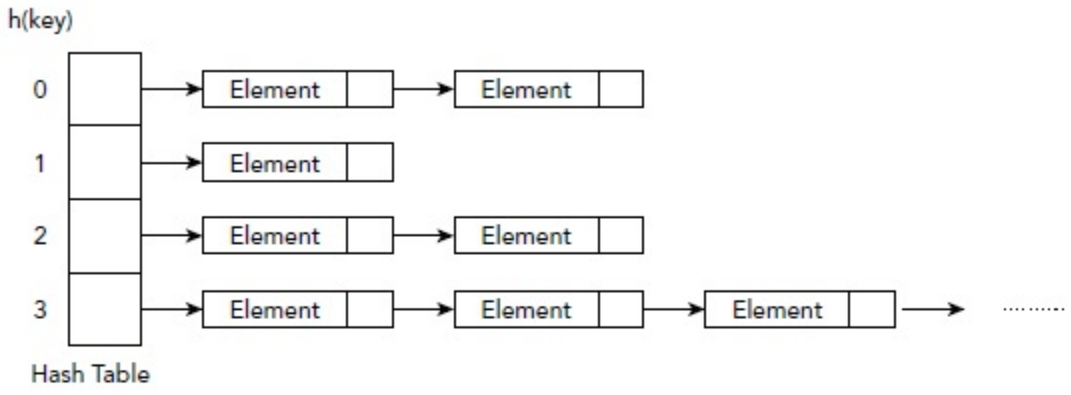
- When analyzing process accounting for Microsoft Windows, you often encounter another linked-list implementation: an embedded doubly linked list.
- We refer to it as "embedded" because it leverages internal storage to embed a _LIST_ENTRY64 data structure within the element being stored.

- **List in Physical and Virtual Memory**

- Dynamic data structures, such as linked lists, are a frequent target of malicious modifications because they can be easily manipulated by simply updating a few links.

h. Hash Tables

- *Hash tables* are often used in circumstances that require efficient insertions and searches where the data being stored is in <key, element> pairs.
- For example, hash tables are used throughout operating systems to store information about active processes, network connections, mounted file systems, and cached files.
- A hash function, h(x), is used to convert the key into an array index, and collisions (i.e., values with the same key) are stored within the linked list associated with the hash table entry.



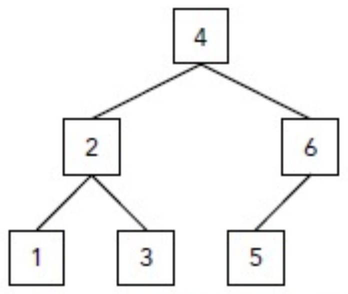
Hash table with chained-overflow example

i. Trees

- A *tree* is another dynamic storage concept that you may encounter when analyzing memory.
- Trees provide a more structured organization of data in memory.

- **Hierarchical Trees**

- A hierarchical tree is composed of a set of nodes used to store elements and a set of links used to connect the nodes.
- Each node also has a key used for ordering the nodes.
- In the case of a hierarchical tree, one node is demarcated as the *root*, and the links between nodes represent the hierarchical structure through parent-child relationships.
- A node that does not have any children (proper descendants) is referred to as a *leaf*. The only non-leaf (internal node) without a parent is the root.



A tree containing six nodes

- **Tree Traversal**

- Especially during memory analysis, is *tree traversal*, which is the process of visiting the nodes of a tree to extract a systematic ordering.
- Each node of a tree can potentially maintain links to multiple children.

- **Analyzing Trees in Memory**

- Analyzing trees in memory also shares similar challenges faced during the analysis of linked lists. For example, physical memory analysis offers the potential to find instances of stored or previously stored elements scattered throughout memory, but does not have the context to discern relationships between those elements.
- On the other hand, you can leverage virtual memory analysis to traverse the tree to extract node relationships and stored elements.
- In the case of an ordered tree, if you know the traversal order or can discern it from the field names, you can extract the ordered list of elements.
- By combining the overall structure of the tree with the ordering criteria, you may even be able to discern information about how the elements were inserted or removed from the tree as it evolved over time.