THE UNIVERSITY OF
WAIKATO
Te Whare Wānanga o Waikato

DEPARTMENT OF
COMPUTER SCIENCE
Te Tari Rorohiko

2024

Contributors

J. Turner

V. Moxham-Bettridge

J. Bowen

INTRODUCTION TO DATA STRUCTURES

Today we will introduce you to the concept of data structures. Data structures are one of the fundamental building blocks of computer science theory and consequently widely used in both computer science and software engineering disciplines. Without data structures computers would not be able to process information quickly and reliably. That is, without data structures almost none of the applications you use everyday would be possible.

What are Data Structures?

Much like the name implies, data structures are ways of organising data so that we can perform operations on that data more efficiently. Examples of common operations include searching for a particular datum (one piece of data) or sorting data from smallest to largest and vice versa.

Depending on the requirements of the software that is being built, it is important to select the correct data structure for your particular situation. Software developers must have a basic conceptual understanding of both the different data structures available as well as their efficiency with respect to time and memory. This allows them to select the correct data structures to process and manage the data they are working with.

There are two types of data structures, linear and non-linear. Linear data structures organise data such that the elements are sequential, that is we can process data one-by-one. In contrast, non-linear data structures do not have any particular sequence but are instead organised in a hierarchical manner based on a defined priority. Today we will look at one example for each of these types of data structures.

Linked Lists: A Linear Data Structure

The official definition of a linked list is as follows:

"An ordered [or unordered] set of data elements, each containing a link to its successor (and sometimes its predecessor)."[5]

[5] Oxford Languages. (n.d.). Linked List Definition. https://www.google.com/search?q=linked+list+definition&rlz=1C5CHFA_enNZ807NZ807&oq=linked+list+definition&aqs=chrome..69i57j35i39l2j69i61l2j69i60l3.2747j0j7&sourceid=chrome&ie=UTF-8

But what does this actually mean? It can be useful to think of data structures in a visual manner in order to understand the underlying concepts. In figure 41 we give an example of a linked list.

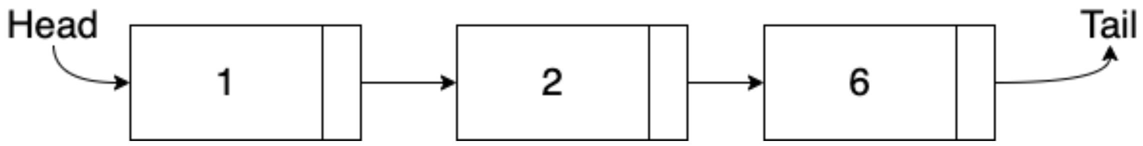

Figure 41: An Example of a Linked List

In figure 41 we have a linked list with the integer data elements “1”, “2” and “6”. Each item in the list is called a “Node” which stores two things: the data value and a link to the next node in the list. We also have a “head” node and a “tail” node. The “head” node points to the very first item in the list while the “tail” node keeps track of the end of the list. However, in most implementations of a linked list we typically only keep track of the head node.

We call a linked list a “self-referential” data structure as each node points to another node i.e. it references an object of its own type. Linked lists have some standard operations that we implement in order to allow us to update the data stored. These operations are: add an item to the list; remove an item from the list; check if the list has a particular value; check if the list is empty; get a specified item from the list; get the total length/size of the list; and so on. We’ll look at some of these operations next.

First open the following website which contains interactive visualisations of the different data structures: https://visualgo.net/en/list. If there is a popup simply dismiss it by clicking anywhere on the greyed out part of the screen. You should see the following screen displayed:

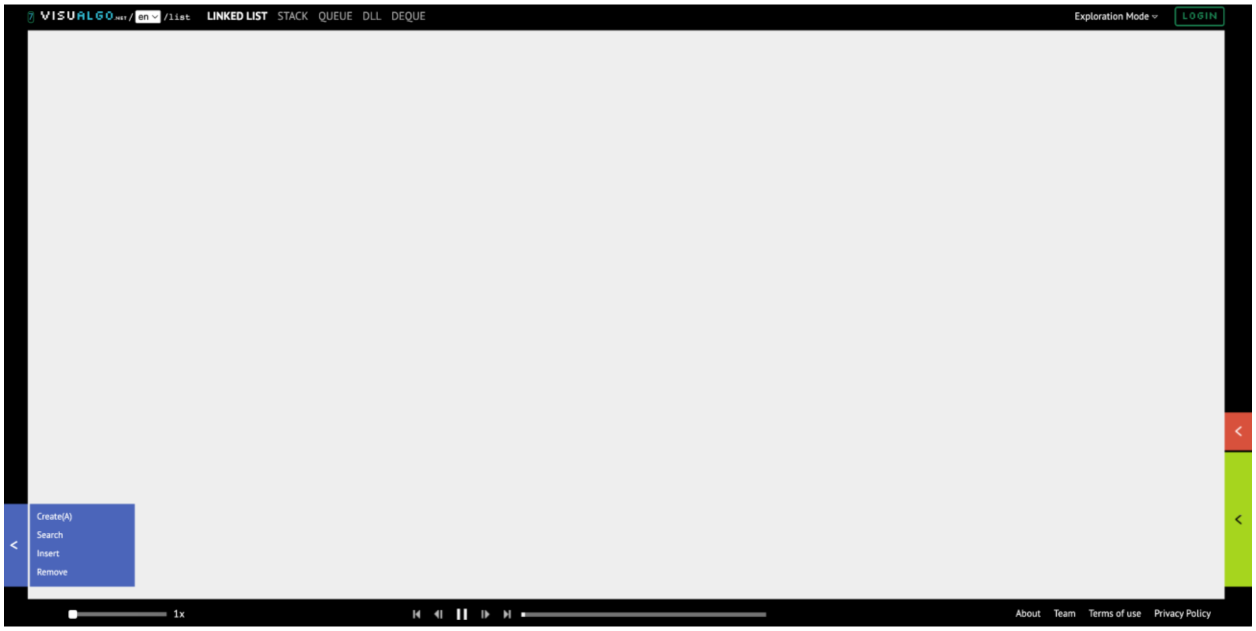

Figure 42: VisuAlgo Linked List

Notice the operations that are available to you in the bottom left hand corner of the screen. You can: create empty (use this to clear the workspace), user defined, random or randomly sorted lists; search for a specific value; insert at the head, tail or at a specific position; and remove at the head, tail or specific position.

Let's start by creating an empty list and we'll explore the different operations. Complete the following exercises:

1. Create an empty list.
2. Insert at the head the values: 45, 77, 10, 1, and 2. Notice how the new node is created and points to the items already in the list? Compare this with adding to the tail. What is the difference?
3. Now try removing the head. Notice how the next item in the list becomes the new head. What happens when you remove the tail? How about an item in the middle of the list?
4. Now create a new linked list with 10 random items, we are going to try the search function next.
5. Try searching for the head, how many nodes did you have to visit?
6. What about an item in the middle of the list, how many nodes do you have to visit then?
7. Try searching for the tail, again how many nodes did you have to visit?

One of the primary issues with a linked list for searching data is that we must always start from the head and search sequentially through the list until we find the item. In terms of Big O complexity (the way we measure how efficient an algorithm is) this means that we must visit, in the worst case when the item is at the tail, all the items of the list in order to find the value. In the best case when the item is at the head, we need to only search 1 item.

Next we will look at Binary Search Trees and how these can be used to improve upon the searching algorithm.

Binary Search Trees: A Non-linear Data Structure

As stated previously, a Binary Search Tree (BST) is a non-linear data structure. Much like with linked lists, we can use BSTs to store data elements and traverse them in sequential order. This is different from a linear structure because of the way the BST stores the data items.

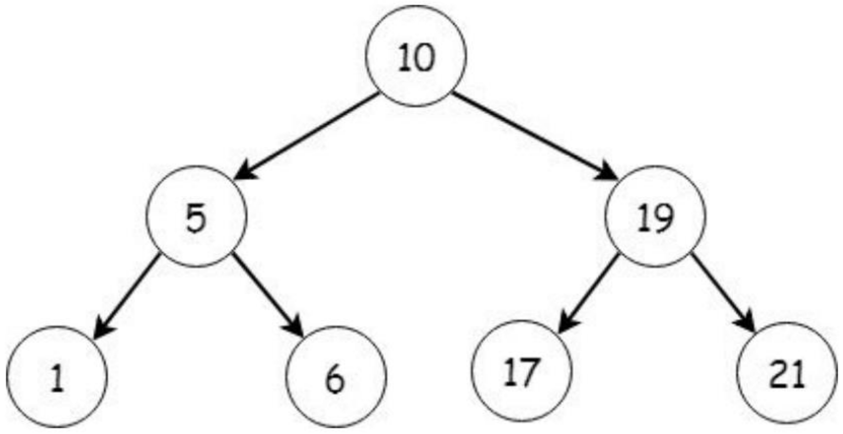

Figure 43: An Example of a BST

Consider Figure 43 where we give an example of a BST. Every node stores a value and two pointers, one to the left node and one to the right node. For any node, the nodes below are usually referred to as children nodes while the node above is referred to as the parent node. For instance, node 10 is the parent of nodes 5 and 19 while 5 and 19 are the children of node 10. Similarly, 17 and 21 are the children of parent node 19.

For any parent node, its left child is always smaller than the parent, meanwhile its right child is always larger than its parent. This is what allows us to maintain the tree order and provide some structure to the tree. This property is quite useful in terms of searching the tree as we can always compare the value and halve the search space each time.

For example, if we are looking for the value "6" in the tree we start with node 10. The value 6 is less than 10 so we move to the left node or rather left subtree. This means we never need to consider any of the values at the right of the tree because they will always be larger than this value. Similarly, when we compare with 5 we can move to the right tree because 6 is larger than

5. Then we have found the value so we can say that 6 is in the tree. In terms of Big O complexity, this search time is referred to as *log(n)* time.

Much like a linked list, a BST is also a self-referential structure. This is because every node points to at most two other nodes (the left and right children). That is, every node is a subtree of the BST as it has the same structure.

Let's look at a visualisation of a BST and explore some of the same operations we did for linked lists: https://visualgo.net/en/bst. You should see the following screen.

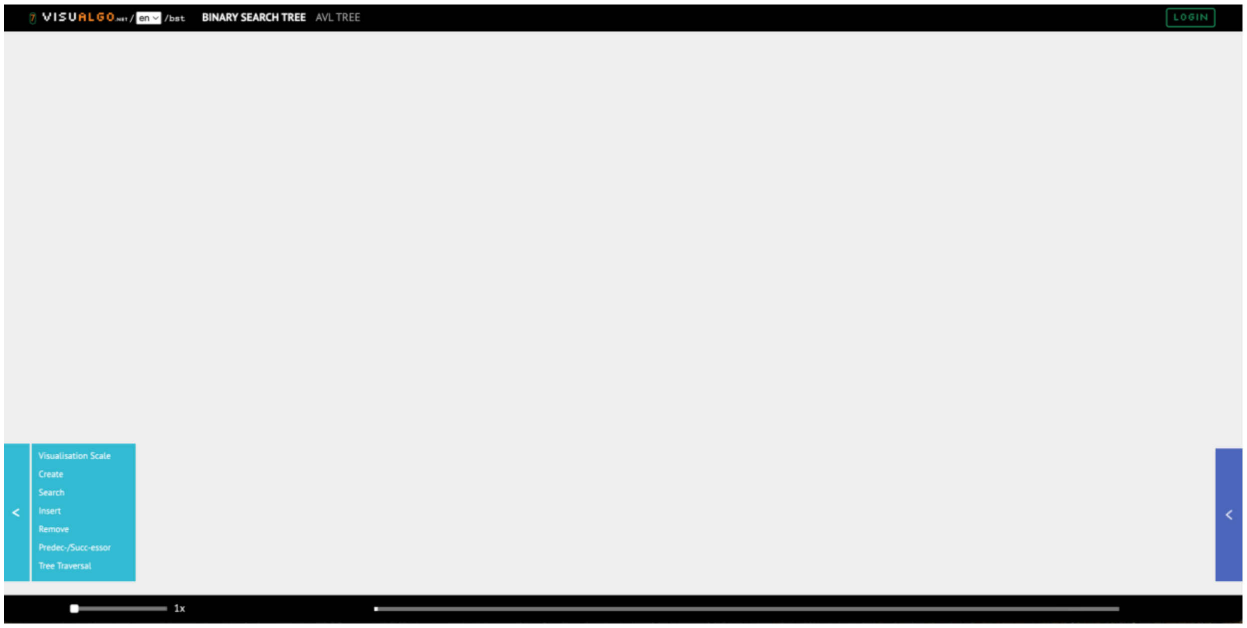

Figure 44: VisuAlgo Binary Search Trees

Complete the following exercises:

1. Create an empty BST.
2. Insert the values: 45, 77, 10, 1 and 2. Now try inserting the same values in a different order (e.g. 77, 1, 45, 2, and 10). How does the structure of the tree change? Why do you think this happens?
3. Try removing the root value (the first item in the tree), what happens? You should see the tree swap the value for the smallest value in the right subtree. This happens because we must maintain the properties of the BST on insertion and deletion.
4. Now create a new BST with a random number of items.
5. Try searching for the root value, how many nodes do you have to visit?
6. What about an item somewhere in the middle of the tree, how many nodes to visit?
7. What about an item at the bottom of the tree (a node with no children), again how many nodes did you have to visit?

The above exercises should help you to better understand the BST operations conceptually. You should notice that we visit fewer nodes when compared with a linked list when searching for data. That is, the ordering of the tree enables faster search times when compared with a linked list.

Implementation

Before moving onto this section ensure that you have a clear understanding of the operations of a Linked List. Explore the visualisation to ensure that you follow the concepts and how the operations work.

In order to implement the data structures given above we need to take advantage of object-oriented programming. We do not give details of this here as it is outside the scope of the session. However, if you are interested in further details you can enrol in our COMPX102 Object-Oriented Programming course with Unistart[6] if you meet the entry requirements.

For simplicity we will focus on implementing a Linked List using the Java programming language. We are going to program this data structure without using an Integrated Development Environment (IDE) like Visual Studio Code or Eclipse. It is useful to learn programming in this way, as in professional development, you are not always guaranteed to be working within a GUI environment (especially when it comes to networking!).

Open up Notepad++ and create a new document called `LinkedList.java`. This is where we will store the details for our data structure. In this example we will create a simple linked list that stores integers.

First we will create an internal Java class called Node which stores the data in the list. It should include the following information:

```
01   class Node {
02        int value;
03        Node next;
04
05        public Node(int x){
06             value = x;
07             next = null;
08        }
09   }
```

[6] See https://www.waikato.ac.nz/study/unistart for more details.

On line 1 we declare the internal Node class. On lines 2-3 we declare the different attributes or rather pieces of data that this class will store. On lines 5-7 we define the constructor method for this class which tells the program how to build a Node when it is created.

Next we create the code for the Linked List, again for simplicity we will focus on an iterative as opposed to recursive solution.

```
10   public class LinkedList {
11        private Node head;
12
13        public void add(int x){
14             Node n = new Node(x);
15             n.next = head;
16             head = n;
17        }
18        …
```

On lines 10-18 we begin defining the linked list implementation. On line 11 we create the head node that we will use as the main entry point into the list. On lines 13-16 we define the method that will add a new item to the linked list. Firstly on line 14 we create the new node with the value of x. On line 15 we make the new node point to the current head of the list, then we set the head to the new node we just created. If you're unsure about this ask a staff member to explain further.

```
18        …
19        public void print(){
20             Node current = head;
21             while(current != null){
22                  System.out.print(current.value + "->");
23                  current = current.next;
24             }
25             System.out.println();
26        }
27        …
```

The print function allows us to print the contents of the list to standard output so that we can see what is in the list. On line 20 we set a current variable that we will use to iterate through the items in the list, because we start from the head it is initialised to this.

On line 21 we start the iteration, we keep moving to the next item in the list as long as it is not null. Null represents the "empty" object in Java. For each item in this list, we print out its value followed by a pointer (`->`). On line 23 we update current to the next item in the list. Lastly, on line 25 we add a new line character to the output.

Next we will create a main method that will be run when the program is executed. This will allow us to debug the code we've written above.

```
27          …
28          public static void main(String[] args){
29               LinkedList myList = new LinkedList();
30               myList.add(4);
31               myList.add(32);
32               myList.add(8);
33               myList.add(16);
34               myList.print();
35          }
36     }
```

This main method creates a new LinkedList object on line 29 and adds 4 values to it (lines 30-33). We then print the output of the list using our print function.

To run this code we need to open the command prompt and navigate to the folder where you stored your java files (ask for assistance if you're not sure how to do this). From there we need to compile the code by using the following command: `javac *.java`. This will compile all the Java files in that folder.

Next we use the following command (still in the command prompt) to run the program and execute the main method: `java LinkedList`. If you have done this correctly you should get the following output: `16->8->31->4`. If you run into any issues, get a staff member to assist you. We will leave the other operations of the Linked List implementation as advanced exercises.

Summary

In this session we introduced data structures, specifically focusing on examples of linear and non-linear structures. We introduced Linked Lists and BSTs and explored the different operations for each. In particular, we looked at how search performance differs between both data types, and how BSTs are more efficient for searching as they halve the search time giving logarithmic search times. Lastly, we finished by beginning an iterative implementation of a Linked List in the Java programming language.

Advanced Exercises

Finish your Linked List implementation in the iterative style. You should have a method for each of the following operations:

- `search()` - search the list for a specific item, return true if the item is in the list, false otherwise.
- `remove(int x)` - remove a specified item from the list.
- `isEmpty()` - returns true if there are no items in the list, false otherwise.
- `length()` - returns the number of nodes in the list.

Useful Resources

- Weiss, M. (2013). Data Structures and Problem Solving using Java : Pearson New International Edition. Pearson Education, Limited.
- Data Structures: Linked Lists: https://www.youtube.com/watch?v=njTh_OwMljA
- Binary search in 4 minutes: https://www.youtube.com/watch?v=fDKIpRe8GW4
- Introduction to Java: https://www.w3schools.com/java/java_intro.asp