

# Go Track

## Week 3 Practical



**NGEE ANN**  
POLYTECHNIC

All rights reserved. No part of this document may be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose, without the express written permission of the author.

Trademarked names may appear in this document. Rather than use a trademark symbol with every occurrence of a trademarked name, the names are used only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The information in this document is distributed on an “as is” basis, without warranty. Although every precaution has been taken in the preparation of this document, the author shall not have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in this document.

## Linked List

### Keywords, Variables and Constants



Description	<i>In this lesson, you will learn about the linked list data structure in Go.</i>
Learning Objectives	<ul style="list-style-type: none"> <li>• Understand the Linked List structure</li> <li>• Able to implement a Linked List</li> <li>• Understand the performance considerations of a Linked List structure</li> </ul>
Duration	40 minutes

### Keywords

A linked list is a commonly used data structure used to store data in a sequential manner. However, it is to be distinguished from arrays, as its implementation usually involves using pointers to link the various elements together and also in the memory, the elements do not reside in consecutive memory space. Thus the superiority of linked list usually lies in its efficient memory usage and its ability to change its size dynamically.

However, with these advantages, it comes accompanied with disadvantages such as the cost of accessing an element in a specific position in the linked list is  $O(n)$ , with the need to traverse to that position; For arrays, similar operation will take only  $O(1)$  constant time. Also additional amount of memory is needed to store the pointer linkages.

Implementation of a linked list involves defining a node and each node will contain item to be stored and next pointer that links to the next node. In this way, several nodes can be linked together to form a list.

	<b>Doubly Linked List</b>	A doubly linked list is a variation of the singly linked list. Each node consists of an item and two pointers, instead of just a next pointer. One pointer points to the previous node while the second pointer points to the next node. This allows better time performance in operations that involve accessing of its elements.
	<b>Circular Linked List</b>	A circular linked list is a variation of linked list. In involves the first node having a pointer to last node, while the last node has a pointer pointing to first node. A circular linked list can be constructed from a singly or doubly linked list. Again, this allows for better performance in accessing of its elements, especially the one at last position.
	<b>Linked list vs Array vs Slice</b>	A linked list is not always a good choice for sequential structure. In situations whereby it is known beforehand the maximum possible number of elements and the total number of elements involved is minimal, an array or slice is a better option.

### Activity #1: Create a simple main program that uses a linked list structure

Download `linkedList.Go`. It contains the base code for a linked list structure.

Create a basic Go main program that makes use of the base code as follows:

1. Create an instance of the linked list.
2. Add 4 items to it by calling the `addNode()` function.
3. Now call the `printAllNodes` function to verify that the items were added correctly to the linked list.

### Activity #2: Adding more functions to the linked list structure

Let's add on to the base code for a linked list structure to introduce other important functions:

1. `remove()` that removes item at a specified position.
2. `addAtPos()` that adds item at a specified position.
3. `get()` that retrieves the item at a specified position.
4. Now write code in your main program to test the above 3 functions.

## Stack

Description	<i>In this lesson, you will learn stack data structure in Go.</i>
Learning Objectives	<ul style="list-style-type: none"> <li>• Understand the Stack structure</li> <li>• Able to implement a Stack</li> </ul>
Duration	30 minutes

A stack is another common data structure used to store data in a Last-In-First-Out (LIFO) manner. Elements can only be added to, and removed from the top of the stack structure.

### Activity #1: Create a simple main program that uses a stack structure

Download stack.Go. It contains the base code for a stack structure.

Create a basic Go main program that makes use of the base code as follows:

1. Create an instance of the stack.
2. Push 4 items it by calling the push() function.
3. Now call the printAllNodes() function to verify the nodes were pushed correctly into the stack. (The items should be printed in the reverse manner that they go into the stack.)

### Activity #2: Printing contents of the stack in main program

Sometimes, there is no function in the stack structure that allows printing of all its nodes. Now you would write code in the main program that will allow you to print the contents in a stack. (hint you may need another instance of certain data structure)

### Activity #3: Checking for unbalanced parenthesis in a program

Stacks can be used to check for unbalanced parenthesis in a program. Write a program that checks a user input of a program and displays if the program contains unbalanced parenthesis.

## Queues


### Keywords, Variables and Constants

Description	<i>In this lesson, you will learn about the queue data structure in Go.</i>
Learning Objectives	<ul style="list-style-type: none"> <li>• Understand the Queue structure</li> <li>• Able to implement a Queue</li> <li>• Understand the performance considerations of a Queue structure</li> </ul>
Duration	<b>40 minutes</b>

### Keywords

A queue is a popular data structure with a distinctive FIFO behaviour. It is being used in several applications and real-life scenarios, that needed to take advantage of FIFO feature. Examples are cpu and disk scheduling, data buffers, queue management, customer relationship management, just to name a few.

Queues can be implemented in a few ways. It can be implemented using arrays, and it can also be implemented using pointers from a modified linked list structure. Items can only be added at the back of a queue and removed from the front of the queue, each of the operations taking constant time  $O(1)$  regardless of how the size of the queue grows. And because in pointer-based queue, elements are not arranged in contiguous memory (unlike array based), memory is being used much more efficiently. However as with linked list, additional memory cost is needed to store the pointer linkages, which is usually a small concern with the memory savings given by the pointer based structure.

	<b>Priority Queue</b>	<p>A priority queue is a modified queue data structure in which each element has a priority associated with it. Elements with higher priority will be processed before elements of lower priority. If they have the same priority level, then they would be handled in the order in which they are enqueued.</p> <p>However, priority queues are also commonly implemented using heaps, which are specialized tree structures.</p>
	<b>Double Ended Queue</b>	<p>A double ended queue is a specialized queue in which items can added or removed both form the front and the back.</p> <p>Its common operations are  <b>push_back:</b> inserts item at back  <b>push_front:</b> inserts item at front  <b>pop_back:</b> removes item from back  <b>pop_front:</b> removes item from front  <b>get_front:</b> returns the first element  <b>get_back:</b> returns the last element</p>

### Activity #1: Create a simple main program that uses a queue structure

Download queue.go. It contains the base code for a queue structure.

Create a basic Go main program that makes use of the base code as follows:

1. Create an instance of the queue.
2. Add 4 items to it by calling the enqueue() function.
3. Now call the printAllNodes function to verify that the items were added correctly to the queue.

### Activity #2: Determining Palindromes

Let's write some code in your driver program (i.e. your main) to determine if an input string is a palindrome. A palindrome is any sequence of characters that reads the same forwards and backwards. E.g. words like "refer", "noon", "level", "my gym", "top spot"

(hint: you need to make use of a queue and stack structure)

### Activity #3: Implementing a Priority Queue

Now improve on queue.go to become a priority queue. As mentioned, elements with higher priority will be processed before elements of lower priority, therefore they will appear in the queue nearer to front compared to those with lower priority. If they are of same priority level, they will be processed in order of their enqueue.

Let higher priority be represented by a smaller number in our case. Larger priority number indicate less importance.

(hint: need to modify enqueue function, as well as node struct of queue to include priority level)

## Tree

Description	<i>In this lesson, you will learn tree data structure in Go.</i>
<b>Learning Objectives</b>	<ul style="list-style-type: none"> <li>• <b>Understand the tree structure</b></li> <li>• <b>Able to implement a tree</b></li> </ul>
<b>Duration</b>	<b>30 minutes</b>

Tree data structure is often used when the data needs to be stored or processed in a hierarchical manner. Possible applications of tree structure are decision trees, expression trees, or cases whereby search for data in the data structure needs to be done efficiently.

We have looked at a particular type of tree structure, namely the Binary Search Tree. Each node in the tree has at the most two children. In addition, the values of the nodes should be sorted. In an almost balanced Binary Search tree, the worse case time complexity for the search of an item is  $O(\log_2 n)$ , which is faster than search for an item sequentially in a linked list. However, in the case if the binary search tree is not balanced, then worse case would be the same as the performance of a linked list, which is  $O(n)$ .

### Activity #1: Create a simple main program that uses a tree structure

Download tree.Go. It contains the base code for a binary search tree structure. Create a basic Go main program that makes use of the base code as follows:

1. Create an instance of the tree.
2. Insert a few items it by calling the insert() function.
3. Now call the inOrder() traversal function to verify the nodes were inserted correctly into the tree.

### Activity #2: Search function for the tree structure

Now implement a search function to search for an item in a binary search tree.

### Activity #3: PreOrder and PostOrder traversals

You have seen how InOrder traversal can be implemented, as provided in the code base. Now implement the other two traversals: PreOrder and PostOrder.

### Activity #4: Count no of nodes function for the tree

Implement a function that counts total number of nodes in the tree.

### Activity #5: Remove function for the tree structure

Now implement a remove function to remove an item from the binary search tree.

## Searching

Description	<i>In this lesson, you will learn about searching algorithms in Go.</i>
<b>Learning Objectives</b>	<ul style="list-style-type: none"> <li>• Understand sequential and binary search algorithms</li> <li>• Implement the search algorithms in Go</li> <li>• Appreciate the performance considerations of search algorithms</li> </ul>
<b>Duration</b>	<b>40 minutes</b>

Searching is one of the important operations that are often needed in a wide range of practical applications.

Searching is the process of finding a particular element in an array or some form of sequence. Typically, a method that implements searching will return the index at which a particular element appears, or -1 if that element does not appear at all. The element you are searching for is called the key or target.

It is important to know how to implement searching algorithms; it is even more important to understand how to use these operations effectively while considering the different constraints and performance efficiencies. Many algorithms exist for searching; choosing and implementing the right one for an application can have a profound effect on how efficiently that application runs.

### Activity #1: Iterative Sequential Search

Write an iterative sequential search function to search for a target in a sorted array of integer numbers. The function header is given below.

```
// search for an item in a sorted array using sequential search (Iterative version)
// returns the index of the item in the array if found, otherwise returns -1
func searchSorted (arr []int, arraySize int, target int) int
```

### Activity #2: Iterative Binary Search

Write an iterative binary search function to search for a target in a sorted array of integer numbers. The function header is given below.

```
// search for an item in a sorted array using binary search (Iterative version)
// returns the index of the item in the array if found, otherwise returns -1
func binarySearch (arr []int, int arraySize, int target) int
```

### Activity #3: Write a program to compare performance of Search algorithms

Modify the search() methods to count the number of comparisons.

Write a go program and do the following in the main function:

1. Declare an integer array, arr, of size 1000.
2. Initialize the arr with even numbers below:  
  
2, 4, 6, . . . , 1996, 1998, 2000
3. Prompt the user to enter a target number to search.
4. Read the number entered and store it to a variable, target
5. Invoke the search() methods above to search for the target and “Found” or “Not Found”



6. Run the sample program with different input numbers and analyze the results:

Target	Sequential Search (Comparisons)	Binary Search (Comparisons)
1		
2		
99		
100		
999		
1000		
1999		
2000		

#### Activity #4: Recursive Sequential Search

Write a recursive sequential search function to search for a target in a sorted array of integer numbers. The function header is given below.

```
// search for an item in a sorted array using sequential search (Recursive version)
// returns the index of the item in the array if found, otherwise returns -1
func searchSortedR(arr []int, n int, start int, target int) int
```

#### Activity #5: Recursive Binary Search

Write a recursive binary search function to search for a target in a sorted array of integer numbers. The function header is given below.

```
// search for an item in a sorted array using binary search (Recursive version)
// returns the index of the item in the array if found, otherwise returns -1
func binarySearchR(arr []int, first int, last int, target int) int
```

## Sorting

Description	<i>In this lesson, you will learn about Sorting Algorithms in Go.</i>
Learning Objectives	<ul style="list-style-type: none"> <li>• Understand sorting algorithms</li> <li>• Implement sorting algorithms in Go</li> <li>• Appreciate the performance considerations of sorting algorithms</li> </ul>
Duration	30 minutes

Sorting is a very classic problem of reordering items of an array (or a list) in a certain order (increasing, non-decreasing, decreasing, non-increasing, lexicographical, etc).

There are many different sorting algorithms, each has its own advantages and limitations. The importance of sorting lies in the fact that data searching can be optimized to a very high level, if data is stored in a sorted manner. Sorting is also used to represent data in more readable formats.

### Activity #1: Performance Comparison between Selection and Insertion Sort

- Trace the insertion sort as it sorts the following array into ascending order

20	80	40	25	60	40
----	----	----	----	----	----

- Trace the selection sort as it sorts the following array into ascending order

7	12	24	4	19	32
---	----	----	---	----	----

- Apply the selection sort and insertion sort to

- An inverted array

8	6	4	2
---	---	---	---

- An ordered array

2	4	6	8
---	---	---	---

What can you conclude about the differences in performance between insertion sort and selection sort based on the order of the items?

### Activity #2: Performance of Merge Sort

- Trace the mergesort algorithm as it sorts the following array into ascending order. List the calls to mergesort and to merge in the order they occur.

20	80	40	25	60	30
----	----	----	----	----	----

- When sorting an array by mergesort,
  - Do the recursive calls to mergesort depend on the values in the array, the number of items in the array, or both. Explain.
  - In what step of mergesort are the items in the array actually swapped (that is sorted)? Explain.
- Discuss the performance of the mergesort algorithm.

### Activity #3: Performance of Quick Sort

- Trace the quicksort as it sorts the following array into ascending order. List the calls to quickSort and to partition in the order in which they occur.

20	80	40	25	60	10	15
----	----	----	----	----	----	----

- You can choose any array item as the pivot for quicksort. Simply interchange items so that your pivot is in theArray[first]. One way to choose a pivot is to take the middle value of the three values theArray[first], theArray[(first + last)/2], and theArray[last]. How many recursive calls are necessary to sort an array of size n if you always choose the pivot in this way?
- Describe the case in which quick sort has the worst performance.

### Activity #4: Advanced Sorting Algorithms

Hybrid sorting algorithms can be created by combining the benefits of various basic sorting algorithms.

Explain how MergeSort algorithm can be improved with Insertion Sort algorithm. Implement the algorithm.