Introduction to Linear Data Structure

A Classification of Data Structures

Data structures can be classified into two main categories: linear and non-linear data structures. Linear data structures are those in which elements are arranged in a sequential manner, allowing for a single level of hierarchy. Examples include arrays, linked lists, stacks, and queues. In contrast, non-linear data structures, such as trees and graphs, allow for multiple levels of hierarchy and do not maintain a sequential arrangement of elements.
The primary difference between linear and non-linear data structures lies in the way their elements are organized and accessed:

- **Linear Data Structures:** Elements are arranged in a sequential order, and each element has a unique predecessor and successor (except the first and last elements). Traversal is typically done in a single run, moving from one element to the next. Examples include arrays, linked lists, stacks, and queues.
- **Non-Linear Data Structures:** Elements are arranged in a hierarchical or interconnected manner, where a single element can be connected to multiple elements. Traversal may require visiting nodes in a non-sequential way. Examples include trees (hierarchical structure) and graphs (network structure).

B Real World Applications of Linear Data Structures

Linear data structures are widely used in various real-world applications due to their simplicity and efficiency. Some common applications include:

- **Arrays:** Used for storing collections of data, such as lists of items, scores, or records in databases. Arrays provide fast access to elements using indices.
- **Linked Lists:** Used in applications where dynamic memory allocation is required, such as implementing queues, stacks, and hash tables. Linked lists allow for efficient insertion and deletion operations.
- **Stacks:** Used in function call management, expression evaluation (e.g., parsing expressions), and backtracking algorithms (e.g., maze solving). Stacks follow the Last In First Out (LIFO) principle.
- **Queues:** Used in scheduling tasks, managing resources (e.g., print jobs), and implementing breadth-first search algorithms. Queues follow the First In First Out (FIFO) principle.

These data structures are fundamental in computer science and are used in various algorithms and applications, including sorting, searching, and data manipulation tasks.

C Memory Representation of Linear Data Structures

Linear data structures can be represented in memory using two primary methods: contiguous memory allocation and linked memory allocation.

C.1 Contiguous Memory Allocation

In contiguous memory allocation, all elements of a linear data structure are stored in consecutive memory locations. This method is commonly used for arrays, where each element can be accessed using its index. The advantages of this method include fast access times and efficient use of memory. However, it can lead to issues such as memory fragmentation and difficulty in resizing the structure.

C.2 Linked Memory Allocation

In linked memory allocation, each element of a linear data structure contains a reference (or pointer) to the next element in the sequence. This method is commonly used for linked lists, stacks, and queues. The advantages of linked memory allocation include dynamic memory usage and ease of insertion and deletion operations. However, it can lead to slower access times due to the need to follow pointers.

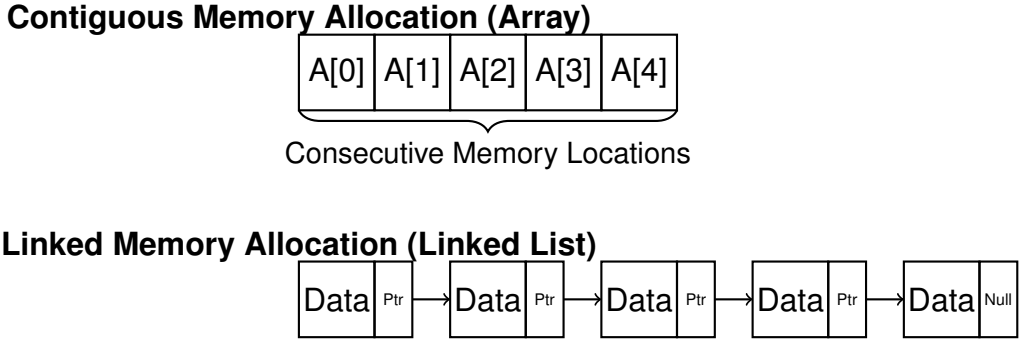

Figure 1: Illustration of contiguous (array) and linked (linked list) memory allocation.

D Types of Linear Data Structures

Linear data structures can be categorized into several types, each with its own characteristics and use cases:

- **Arrays:** A collection of elements stored in contiguous memory locations, allowing for efficient access using indices. Arrays have a fixed size and are suitable for storing homogeneous data.
- **Linked Lists:** A sequence of elements where each element (node) contains a value and a reference to the next node. Linked lists allow for dynamic resizing and efficient insertion and deletion operations.
- **Stacks:** A linear data structure that follows the Last In First Out (LIFO) principle, where elements can be added or removed only from the top.
- **Queues:** A linear data structure that follows the First In First Out (FIFO) principle, where elements are added at the rear and removed from the front.

E Pointers and Dynamic Memory Allocation

Pointers are variables that store memory addresses of other variables. They are a fundamental concept in programming languages like C and C++, allowing for dynamic memory management and efficient data structure manipulation. Pointers enable the creation of complex data structures such as linked lists, trees, and graphs by linking nodes together through their addresses. Pointers can be used to access and modify data stored in memory, enabling efficient algorithms and data manipulation. They are particularly useful in scenarios where dynamic memory allocation is required, such as when the size of a data structure is not known at compile time.

Example

```
int a = 10; // Declare an integer variable
int *p; // Declare a pointer to an integer
p = &a; // Assign the address of 'a' to the pointer 'p'

printf("Value of a: %d\n", a); // Output: 10
printf("Address of a: %p\n", (void*)&a); // Output: Address of 'a'
printf("Value pointed by p: %d\n", *p); // Output: 10
printf("Address stored in p: %p\n", (void*)p); // Output: Address
   of 'a'
```

In this example, we declare an integer variable `a` and a pointer `p`. We assign the address of `a` to `p`, allowing us to access the value of `a` through the pointer. The `*` operator is used to dereference the pointer, retrieving the value stored at the address it points to.

E.1 Pointer Arithmetic

Pointer arithmetic refers to the operations that can be performed on pointers to navigate through memory locations. In languages like C and C++, pointers can be incremented or decremented to move to the next or previous memory location, respectively. This is particularly useful when working with arrays, as it allows for efficient traversal and manipulation of elements.

Example

```
int arr[] = {10, 20, 30, 40, 50}; // Declare an array
int *p = arr; // Assign the address of the first element to the
   pointer

for (int i = 0; i < 5; i++) {
    printf("Element %d: %d\n", i, *(p + i)); // Access elements
       using pointer arithmetic
}
```

In this example, we declare an array `arr` and assign the address of its first element to the pointer `p`. We then use pointer arithmetic to access each element of the array by incrementing the pointer `p`.

E.2 Pointer to Pointer

A pointer to a pointer is a variable that stores the address of another pointer. This concept is useful in scenarios where we need to manipulate pointers dynamically, such as in multi-dimensional arrays or when passing pointers to functions.

Example

```
int a = 10; // Declare an integer variable
int *p = &a; // Declare a pointer to 'a'
int **pp = &p; // Declare a pointer to the pointer 'p'

printf("Value of a: %d\n", a); // Output: 10
printf("Value pointed by p: %d\n", *p); // Output: 10
printf("Value pointed by pp: %d\n", **pp); // Output: 10
```

In this example, we declare an integer variable `a`, a pointer `p` that points to `a`, and a pointer to a pointer `pp` that points to `p`. We can access the value of `a` through both `p` and `pp`.

E.3 Dynamic Memory Allocation

Dynamic memory allocation allows programs to request memory at runtime, enabling the creation of data structures whose size can change during execution. In languages like C++, dynamic memory allocation is typically done using functions like `new` and `delete`. This is particularly useful for implementing data structures like linked lists, trees, and graphs, where the size may not be known at compile time.

Example

```
int *p = new int; // Dynamically allocate memory for a single
   integer
if (p == nullptr) {
    std::cout << "Memory␣allocation␣failed" << std::endl;
    return 1; // Exit if memory allocation fails
}
*p = 42; // Assign a value to the allocated memory
std::cout << "Value:␣" << *p << std::endl; // Output: 42
delete p; // Free the allocated memory
```

In this example, we dynamically allocate memory for a single integer using the `new` operator. We assign a value to the allocated memory, output it, and then free the memory using `delete`.

Flow of Dynamic Memory Allocation:

- When `new int` is called, the program requests a block of memory large enough to store an integer from the heap (free store).
- The memory manager checks if there is enough free space available in the heap.
- If sufficient space exists, it reserves (allocates) a block of memory and marks it as in use.
- The address of this block is returned as a pointer (`arr`) to the program.
- If there is not enough free space, `new` returns `nullptr` to indicate failure.
- When `delete` is called, the memory manager marks the previously allocated block as free, making it available for future allocations.

F Arrays

An array is a collection of elements stored in contiguous memory locations, allowing for efficient access using indices. Arrays can be one-dimensional (like a list) or multi-dimensional (like a matrix). They are commonly used to store homogeneous data types, such as integers, characters, or floating-point numbers.

F.1 One Dimensional Arrays

One-dimensional arrays are the simplest form of arrays, where elements are stored in a single linear sequence. Each element can be accessed using its index, which starts from 0. One-dimensional arrays are useful for storing lists of items, such as scores, names, or any other homogeneous data.

Example

```
int *arr = new int[5]; // Allocate memory for an array of 5
   integers
if (arr == nullptr) {
    printf("Memory allocation failed\n");
    return 1; // Exit if memory allocation fails
}

for (int i = 0; i < 5; i++) {
    arr[i] = i * 10; // Initialize the array
}

for (int i = 0; i < 5; i++) {
    printf("Element %d: %d\n", i, arr[i]); // Output the elements
}

delete[] arr; // Free the allocated memory
```

In this example, we dynamically allocate memory for an array of 5 integers using the `new` operator. We check if the memory allocation was successful by verifying that the pointer `arr` is not `nullptr`. After initializing the array, we output its elements and finally free the allocated memory using `delete[]`.
We can also use pointer arithmetic to access the elements of the dynamically allocated array:

Example

```
int *arr = new int[5]; // Allocate memory for an array of 5
   integers
if (arr == nullptr) {
    printf("Memory allocation failed\n");
    return 1; // Exit if memory allocation fails
}

for (int i = 0; i < 5; i++) {
    *(arr + i) = i * 10; // Initialize the array using pointer
       arithmetic
}

for (int i = 0; i < 5; i++) {
    printf("Element %d: %d\n", i, *(arr + i)); // Output the
       elements using pointer arithmetic
}

delete[] arr; // Free the allocated memory
```

In this example, we use pointer arithmetic to initialize and access the elements of the dynamically allocated array. The expression `*(arr + i)` dereferences the pointer to access the value at the `i`-th index.
In this case, the operator `new` returns a pointer to the first element of the allocated memory block. If the allocation fails, it returns `nullptr`. The successive memmory locations for the array elements are determined by the size of the data type being allocated. For example, if we allocate an array of integers, each element will occupy `sizeof(int)` bytes in memory. This memory is contiguous, meaning that all elements are stored in a single block of memory,

allowing for efficient access and manipulation.

F.2 Multi-Dimensional Arrays

Multi-dimensional arrays are arrays that have more than one dimension, allowing for the representation of matrices or higher-dimensional data structures. In C++, multi-dimensional arrays can be implemented using pointers to pointers or by allocating a contiguous block of memory.

To allocate array of two dimension, we can use a pointer to a pointer:

Example

```
int rows = 3, cols = 4;
int **matrix = new int*[rows]; // Allocate memory for an array of
   pointers
for (int i = 0; i < rows; i++) {
    matrix[i] = new int[cols]; // Allocate memory for each row
}
for (int i = 0; i < rows; i++) {
    for (int j = 0; j < cols; j++) {
        matrix[i][j] = i * cols + j; // Initialize the matrix
    }
}
for (int i = 0; i < rows; i++) {
    for (int j = 0; j < cols; j++) {
        printf("%d␣", matrix[i][j]); // Output the elements
    }
    printf("\n");
}
for (int i = 0; i < rows; i++) {
    delete[] matrix[i]; // Free each row
}
delete[] matrix; // Free the array of pointers
```

In this example, we dynamically allocate a 2D array (matrix) using a pointer to a pointer. We first allocate memory for an array of pointers, and then for each row, we allocate memory for the columns. After initializing the matrix, we output its elements and finally free the allocated memory.

Similarly, for a 3D array, we can use a pointer to a pointer to a pointer, for 4D array, we can use a pointer to a pointer to a pointer to a pointer, and so on. The general pattern is to allocate memory for each dimension sequentially. To free the memory allocated for multi-dimensional arrays, we need to free each level of pointers in reverse order of allocation.

F.3 Static vs Dynamic Arrays

Static arrays have a fixed size determined at compile time, while dynamic arrays can change size during runtime. Static arrays are allocated on the stack, leading to faster access times but limited flexibility. Dynamic arrays, allocated on the heap, allow for resizing and more complex data structures but may incur overhead due to memory management.

Example

```
int staticArray[5] = {1, 2, 3, 4, 5}; // Static array with fixed
   size
int *dynamicArray = new int[5]; // Dynamic array with size
   determined at runtime
if (dynamicArray == nullptr) {
    std::cout << "Memory␣allocation␣failed" << std::endl;
    return 1; // Exit if memory allocation fails
}
for (int i = 0; i < 5; i++) {
    dynamicArray[i] = i + 1; // Initialize the dynamic array
}
for (int i = 0; i < 5; i++) {
    std::cout << "Static␣Array␣Element␣" << i << ":␣" <<
       staticArray[i] << std::endl; // Output static array elements
    std::cout << "Dynamic␣Array␣Element␣" << i << ":␣" <<
       dynamicArray[i] << std::endl; // Output dynamic array
       elements
}
delete[] dynamicArray; // Free the allocated memory for dynamic
   array
```

In this example, we declare a static array with a fixed size and a dynamic array whose size is determined at runtime. We initialize both arrays and output their elements. Finally, we free the memory allocated for the dynamic array.

G Common Errors in Linear Data Structures

When working with linear data structures, several common errors can occur, leading to unexpected behavior or crashes. Understanding these errors is crucial for debugging and ensuring the reliability of your code.

G.1 Out of Bounds Access

Out of bounds access occurs when a program attempts to read or write data outside the valid range of an array or other linear data structure. This can lead to undefined behavior, crashes, or data corruption. It is essential to always check the bounds of an array before accessing its elements.

Example

```
int arr[5] = {1, 2, 3, 4, 5};
for (int i = 0; i <= 5; i++) { // Incorrect condition, should be i
   < 5
    std::cout << arr[i] << "␣"; // This will cause out of bounds
       access on the last iteration
}
```

In this example, the loop condition is incorrect, allowing the program to access an element outside the bounds of the array `arr`. The correct condition should be `i < 5` to prevent out of bounds access.

G.2 Null Pointer Dereference

Null pointer dereference occurs when a program attempts to access or modify data through a pointer that has not been initialized or has been set to `nullptr`. This can lead to crashes or undefined behavior. Always ensure that pointers are properly initialized before use.

Example

```
int *p = nullptr; // Pointer is initialized to nullptr
std::cout << *p; // Attempting to dereference a null pointer will
   cause a crash
```

In this example, the pointer `p` is initialized to `nullptr`, and attempting to dereference it will lead to a crash. To avoid this error, always check if a pointer is `nullptr` before dereferencing it.

G.3 Memory Leaks

Memory leaks occur when dynamically allocated memory is not properly deallocated, leading to a gradual increase in memory usage over time. This can happen if a program allocates memory but fails to release it when it is no longer needed. Memory leaks can lead to performance degradation and eventually exhaust available memory, causing the program to crash. To prevent memory leaks, it is essential to always free dynamically allocated memory using the appropriate deallocation functions (e.g., `delete` in C++ or `free` in C) when it is no longer needed. Additionally, tools like Valgrind can help detect memory leaks during development.

Example

```
for (int i = 0; i < 1000; i++) {
    int *arr = new int[1000000]; // Allocate a large array in each
       iteration
    // Process the array (omitted)
    // Forget to free the allocated memory
    // delete[] arr; // Uncommenting this line would prevent the
       memory leak
}
```

In this example, we allocate a large array in each iteration of a loop without freeing the memory afterward. This leads to a memory leak, as the allocated memory is not released, causing the program's memory usage to grow with each iteration. This can eventually lead to performance issues or crashes if the program runs out of memory. To prevent this, we should always ensure that we free the allocated memory when it is no longer needed, as shown in the commented line.

When deallocating memory in C++, it is important to match the allocation and deallocation operators. Use `delete` to free memory allocated for a single object, and `delete[]` to free memory allocated for an array of objects. Using the wrong form can result in undefined behavior.

G.4 Dangling Pointers

Dangling pointers occur when a pointer continues to reference memory that has been deallocated or released. This can happen if a pointer is used after the memory it points to has been freed, leading to undefined behavior or crashes. To avoid dangling pointers, always set pointers to `nullptr` after freeing the memory they point to.

Example

```
int *p = new int(10); // Dynamically allocate memory
delete p; // Free the allocated memory
std::cout << *p; // Attempting to dereference a dangling pointer
   will cause undefined behavior
p = nullptr; // Set the pointer to nullptr to avoid dangling
   pointer
```

In this example, we dynamically allocate memory for an integer and then free it. However, we attempt to dereference the pointer `p` after it has been freed, leading to undefined behavior. To prevent this, we set `p` to `nullptr` after freeing the memory, ensuring that it no longer points to a deallocated memory location.

G.5 Incorrect Pointer Arithmetic

Incorrect pointer arithmetic can lead to accessing invalid memory locations or causing out of bounds access. Pointer arithmetic should be performed carefully, considering the size of the data type being pointed to.

Example

```
int arr[5] = {1, 2, 3, 4, 5};
int *p = arr; // Pointer to the first element of the array
for (int i = 0; i < 5; i++) {
    std::cout << *(p + i) << "␣"; // Correct pointer arithmetic
}
std::cout << *(p + 5) << "␣"; // Incorrect pointer arithmetic,
   accessing out of bounds
```

In this example, we correctly use pointer arithmetic to access the elements of the array `arr`. However, the last line attempts to access an element outside the bounds of the array, leading to undefined behavior. To avoid this error, always ensure that pointer arithmetic stays within the valid range of the data structure.

G.6 Type Mismatch

Type mismatch occurs when a pointer is used to point to a data type that is incompatible with the type of data it is intended to reference. This can lead to incorrect data interpretation and undefined behavior. Always ensure that pointers are correctly typed and match the data they point to.

Example

```
int a = 10;
float *p = (float*)&a; // Incorrect type, pointer to float should
   not point to an int
std::cout << *p; // Attempting to dereference a pointer of
   incorrect type will cause undefined behavior
```

In this example, we declare an integer variable `a` and a pointer `p` of type `float`. Attempting to dereference `p` will lead to undefined behavior because `p` is not correctly typed to point to an integer. To avoid type mismatch errors, always ensure that pointers are declared with the correct data type.

G.7 Memory Fragmentation

Memory fragmentation occurs when free memory is divided into small, non-contiguous blocks, making it difficult to allocate larger blocks of memory. This can happen when memory is frequently allocated and deallocated in varying sizes, leading to inefficient memory usage. To mitigate memory fragmentation, consider using memory pools or custom allocators that manage memory more efficiently.

Example

```
int *p1 = new int[100]; // Allocate 100 integers
delete[] p1; // Free the allocated memory
int *p2 = new int[50]; // Allocate 50 integers
// If the memory allocator cannot find a contiguous block of
   memory for p2, it may lead to fragmentation
```

In this example, we allocate memory for 100 integers and then free it. Next, we attempt to allocate memory for 50 integers. If the memory allocator cannot find a contiguous block of memory for the second allocation due to fragmentation, it may lead to inefficient memory usage or allocation failures. To avoid memory fragmentation, consider using custom memory management techniques that optimize allocation patterns.

Practice Exercise:

Create a class called ‘DynamicArray‘ that manages dynamic memory allocation for an integer array. The class should allocate memory for the array in its constructor and deallocate the memory in its destructor. Implement member functions to set and get elements of the array. The member functions should check for out of bounds access and throw exceptions if an invalid index is used. Demonstrate the usage of the class by creating an object, setting values, retrieving them, and showing that memory is properly released when the object goes out of scope.