Gabriel Opiyo Nyawade

SCT211-0029/2024

B.Sc. Computer Science

ICS 2105 Assignment

Answers:

1. Find largest element in integer array

```
// C++
#include <iostream>
#include <vector>
int max_val(std::vector<int>& array)
{
	int val = array[0];
	for (int i = 1; i < int(array.size()); i++)
	{
		if (array[i] > val)
		val = array[i];
	}
	return val;
}
```

2. Shift zeros to the end while preserving order of non-zero elements

```
// C++
#include <iostream>
#include <vector>

void shift_zeros(std::vector<int>& array)
{
	int temp;
	for (int i = 0; i < int(array.size()); i++)
	{
		for (int j = 1; j < int(array.size()); j++)
		{
			if (array[j] == 0)
			{
				temp = array[j];
				array[j] = array[j + 1];
				array[j + 1] = temp;
			}
		}
	}
}
```

3. Remove duplicates from an integer array

```
// C++
#include <iostream>
#include <vector>
using namespace std;

void remove_duplicates(vector<int>& array)
{
	vector<int> output;
	int val = array[0];
	output.push_back(val);
	for (int i = 1; i < array.size(); i++)
	{
		if (array[i] != val)
		{
			val = array[i];
			output.push_back(val);
		}
	}
	array = output;
}
```

4. Reversal of a singly linked list

```
// C++
#include <iostream>
struct Node
{
      int number;
      Node* next;
};

Node* start = nullptr;

void reverse_list()
{
      Node* ptr = start;
      Node* prev = start;
      ptr = ptr->next;
      prev->next = nullptr;
      while (ptr != nullptr)
      {
            prev = ptr;
            ptr = ptr->next;
            prev->next = start;
            start = prev;
      }
}
```

5. Binary Search on a sorted integer array

```
// C++
#include <iostream>
#include <vector>

void binarySearch(int search_elem, std::vector<int>& elems)
{
	int first, middle, last;
	first = 0;
	last = elems.size() - 1;
	middle = (first + last) / 2;
	while (first <= last)
	{
		if (search_elem == elems[middle])
		{
			std::cout << "Element has been found in index " <<
			middle << " of ";
			display_vector(elems);
			break;
		}
		else if (search_elem > elems[middle])
			first = middle + 1;
		else if (search_elem < elems[middle])
			last = middle - 1;


		middle = (first + last) / 2;
	}
	if (first > last)
	std::cout << "Element has not been found" << "\n";
}
```

6. Depth-first Search Implementation for a Graph

For depth-first search, a stack is used to load the starting vertex, then after processing loads all other neighbours of the vertex, and when the next vertex, is processed, that in turn loads its neighbouring vertices, and this happens recursively until a vertex with no neighbours, or whose neighbours have already been processed is encountered, then it collapses back to the next intermediate vertex that was placed on the stack, so it uncovers the whole depth of a particular path before traversing the next path.

For the implementation, each vertex has its processing state, either `READY` or `WAITING` or `PROCESSED`, and is loaded onto the stack. The algorithm is as follows:

```
STEP 1: SET STATE = 1 (READY) FOR EVERY VERTEX IN G

STEP 2: PUSH STARTING VERTEX ONTO STACK, SET STATE = 2 (WAITING)

STEP 3: REPEAT STEPS 4 AND 5 UNTIL STACK IS EMPTY

      STEP 4: POP THE TOP VERTEX V, PROCESS AND SET STATE = 3
      (PROCESSED)

      STEP 5: PUSH TO STACK ALL NEIGHBOURS OF V THAT HAVE STATE = 1
      AND SET STATUS = 2

[END OF LOOP]

STEP 6: EXIT
```

7. Queue Implementation using two Stacks

Here we have two stacks, say `s1` and `s2`, which we can assume to be unlimited in capacity. Enqueue operations are done in `s1`, where pre-existing entries are popped and pushed into `s2`, causing them to be arranged in reverse order, the last entry being on top. Then the enqueued value is pushed into `s1` at the very bottom, and the entries pushed into `s2` are returned back into `s1`, with the order being the first entry on top. This makes the dequeue operation easier, just popping the top value from `s1`.

```
// C++

class Queue

{

private:

      Stack s1, s2;
```

```
public:
      void enqueue(int val)
      {
            while(!s1.empty())
            {
                  s2.push(s1.top());
                  s1.pop();
            }
            s1.push(val);
            while(!s2.empty())
            {
                  s1.push(s2.top());
                  s2.pop();
            }
      }
      int dequeue()
      {
            if (s1.empty()) return -1;
            int val = s1.top();
            s1.pop();
            return val;
      }
};
```

8. Reversal of a Queue

```
#include <iostream>
#define MAX 10
class Queue
{
private:
      int line[MAX];
      int FRONT;
      int REAR;
public:
      void init()
      {
            FRONT = -1;
            REAR = -1;
      }
      void enqueue(int);
      int dequeue(void);
      void reverse(void);
      void display(void);
};
void Queue::reverse()
{
      int temp;
      while (FRONT - REAR < 0)
      {
            temp = line[REAR];
            line[REAR] = line[FRONT];
            line[FRONT] = temp;
            FRONT++;
            REAR--;
      }
```

```
}
```

9. Arrays

Arrays are data structures that allow for elements of a similar type to be stored contiguously in memory. That means that the memory locations within the array are adjacent to each other, with the first element being addressed by a base address, and all other subsequent elements addressed by an address value equivalent to the base address plus the total size of the elements prior to each, hence array indexing is done in constant time. It does not include traversal through each element, but does a single calculation to obtain the memory address of the indexed value as follows:

```
INDEXED_ADDRESS = BASE_ADDRESS + (I x SIZE_OF_ELEMENT)
```

10. Limitations of Arrays and Dynamic Alternative

The main limitation of an array is that it's size must be known at compile time. Another possible limitation is that it might have redundant storage locations in a case where the elements stored do not exhaust the available locations therein. This poses significant difficulty to the programmer who wishes to implement a dynamic data structure that expands contiguously.

Dynamic alternatives include heap-allocated arrays, where a memory block from the heap is split into contiguous portions of the size of the element being occupied within it, and a pointer to the first element is returned. Here the size of the array can be grown or shrunk at will, and this is by virtue of run-time size calculation and reallocation of memory for the array, whereby the distinction between size and capacity plays a huge role. The size of a dynamic (heap-allocated) array is the number of elements within it, and the capacity is like the maximum size the array can actualise before having to grow by duplication, or vice versa the minimum size it can take, by halving the capacity whenever the size falls below half the current capacity.

11. Structure of a Singly Linked List and its Advantages over Arrays

The singly linked list is a data structure of sequential but non-contiguous data members that "link" or connect to each other via pointers. These data members are referred to as nodes, and are singular memory locations bearing numerous variables grouped together, in this case some data types and a pointer to other nodes of the same type. Principally, there must be an initial pointer that points to the first node, which makes the program able to determine the memory location of the first node, and the last node should not reference anything, it should be rendered null to signify the end of the list. It's data members are accessed sequentially, and indexing is done iteratively, as opposed to arrays that do the same in constant time.

Its advantages over arrays is that it is dynamic, more nodes can be added to it, or nodes removed, at run-time, meaning that insertion and deletion operations are quite efficient. Furthermore, it is much more simpler to implement other data structures such as stacks and queues from linked lists as foundations, rather than from arrays which are static and rigid data structures.

12. Advantages and Disadvantages of Binary Search over Linear Search

Its advantages are:

- Searching takes less time, as the time complexity for binary search is O(log n) as opposed to linear search which takes O(n) time.

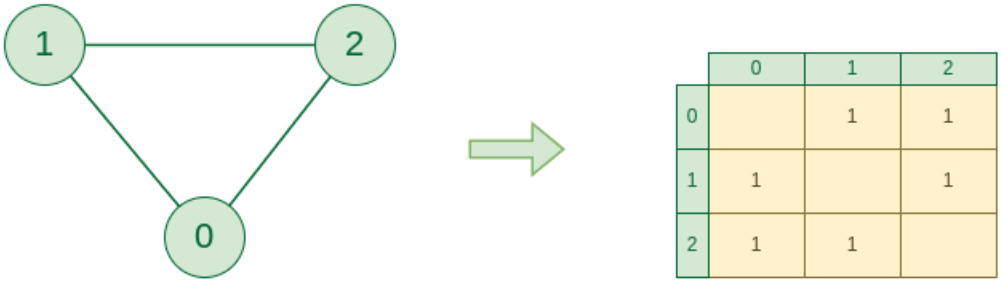
Its disadvantages are:

- It requires that the array in which searching takes place is sorted, thus needing a sorting algorithm in the case of unsorted arrays.
- It only works for one-dimensional arrays.

13. Representation of Graphs in Memory

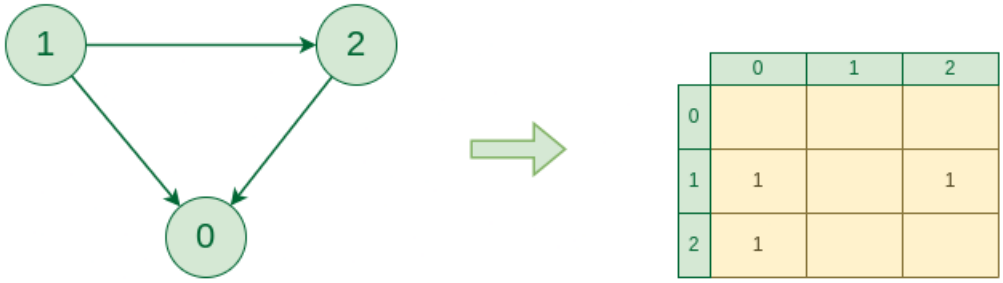
There are two main ways in which graphs can be represented in memory: adjacency matrix, and adjacency list representation.

Adjacency matrix representation: This utilises a two-dimensional array, initialised with a size of n * n, where n is the number of vertices in the graph. Within, it contains either a value zero, representing no edge between two vertices, where the row and column represent the initial vertices and the terminal vertices respectively, a value one, representing a edge between the vertices, and a number greater than one representing a weighted edge between the vertices, the weight equivalent to the number representing the edge. Below are examples for each (courtesy https://www.geeksforgeeks.org/graph-and-its-representations/).



Undirected Graph | Adjacency Matrix

Graph Representation of Undirected graph to Adjacency Matrix

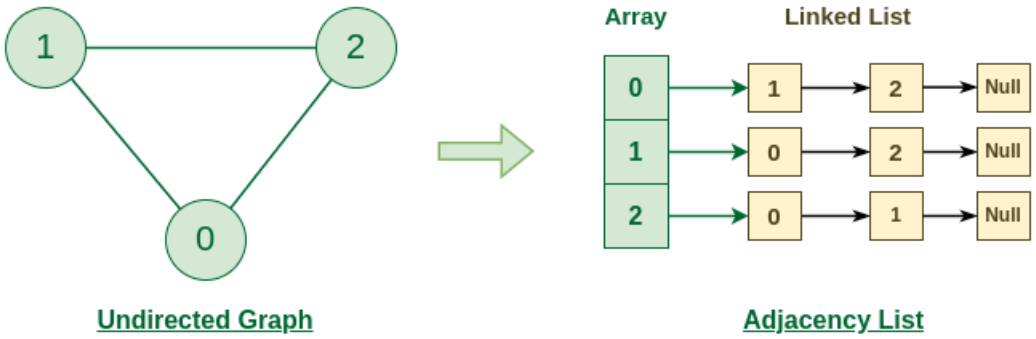


Directed Graph | Adjacency Matrix

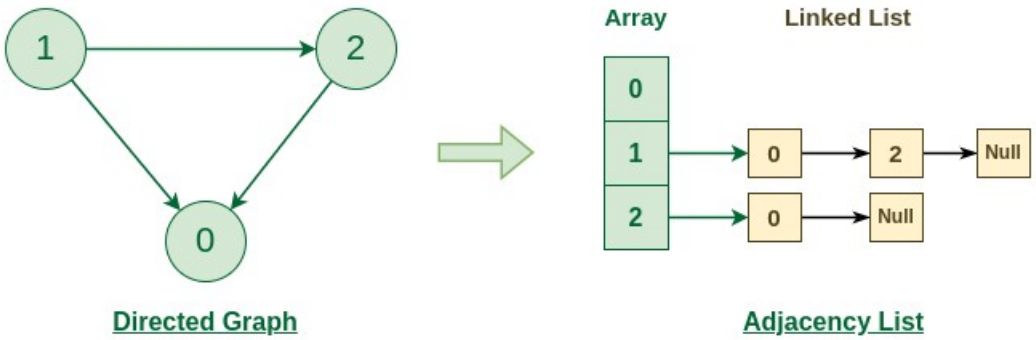
Graph Representation of Directed graph to Adjacency Matrix

Adjacency list representation: This utilises a one-dimensional array with each index representing a vertex of the graph, and containing linked lists, in this

case pointers which serve as head references to each linked list, with each node in the linked list representing a vertex adjacent to the indexed vertex of the array. They also vary depending on whether the graph is directed or undirected. Below are examples for each (courtesy https://www.geeksforgeeks.org/graph-and-its-representations/).



Graph Representation of Undirected graph to Adjacency List



Graph Representation of Directed graph to Adjacency List

14. FIFO Property of Queues and Real Life Analogy

The queue utilises the principle of First-In First-Out (FIFO) to perform insertion and deletion operations. The queue initially has a fixed size, and elements are added towards the end of the array, so that the first element enters in the first position, the second element in the second position, and likewise until the queue is full. For removing elements, it is done initially from the start, progressing towards the end of the array, until the queue is empty. This ensures that the first element inserted is the first to be deleted as well.

Comparing this to a real world scenario, take for instance a queue in a supermarket checkout. People line up, starting from the first one to get to the checkout terminal or cashier, and the first one is always served first then removed from the line, and new people are added at the back of the line.

15. Sparse Matrices and their Representation using Linked Lists

A sparse matrix is a matrix that has most of its values as zero, containing only a few non-zero entries. It can be represented using linked lists whereby each node has the value and position of each non-zero entry, that is, the number, row and column data. Below is an example (courtesy https://www.geeksforgeeks.org/sparse-matrix-representation/).