CP- EITC

Lecture 2

CP Cell Leads:
Idriss EL ABIDI
Souhail LOUZI

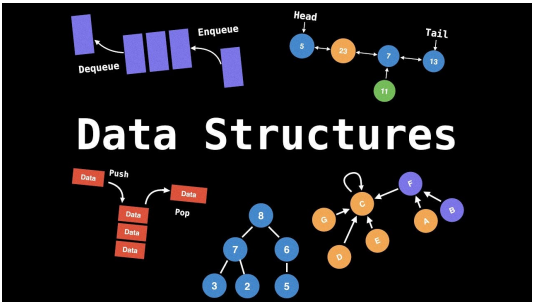
Introduction to Data Structures & C++

December, 2024

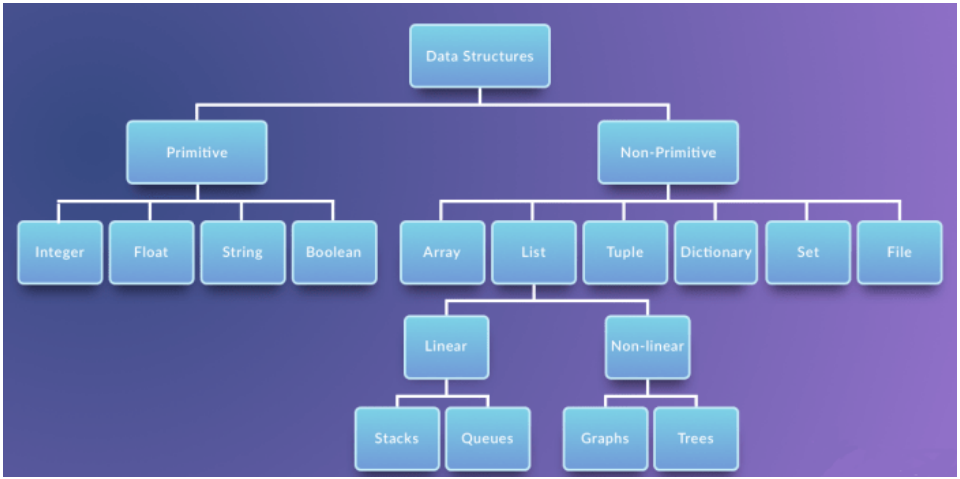
Intoduction :

A data structure is a way to store data in the memory of a computer. It is important to choose an appropriate data structure for a problem, because each data structure has its own advantages and disadvantages.

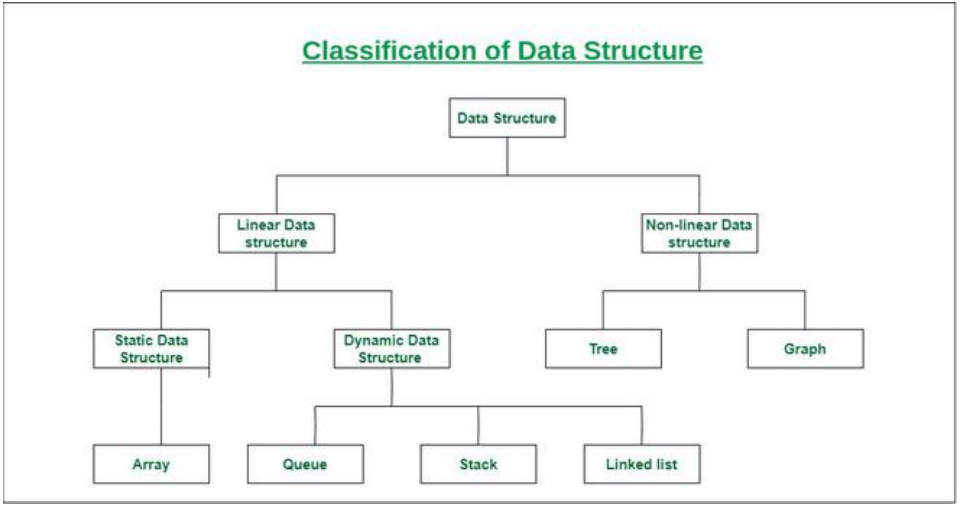
The crucial question is: which operations are efficient in the chosen data structure?



Classification/Types of Data Structures:

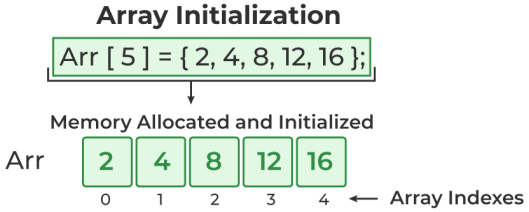


Classification/Types of Data Structures:



Definiton:

An array is a collection of data items stored at contiguous memory locations. The idea is to store multiple items of the same type together. This makes it easier to calculate the position of each element by simply adding an offset to a base value, i.e., the memory location of the first element of the array (generally denoted by the name of the array).



Basic Operations

Array

- **declaration:** an array should be declared first using the size of the array in C++ :
  int arr[17];
- **accessing:** we access each element of index i by using arr[i];

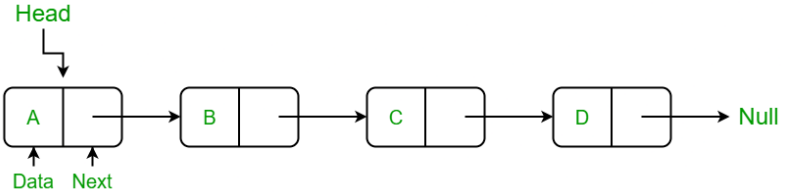
Matrix/Grid

- **declaration:** an array should be declared first using the size of each dimension of the array in C++ :
  int arr[17][18];
- **accessing:** we access each element of index i,j by using arr[i][j];

Linked Lists

Definiton:

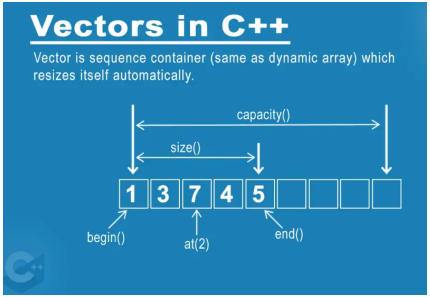
Like arrays, Linked List is a linear data structure. Unlike arrays, linked list elements are not stored at a contiguous location; the elements are linked using pointers.



Dynamic Arrays - Vectors

Definition :

Dynamic arrays (vector in C++) support all the functions that a normal array does, and can resize itself to accommodate more elements. Vectors in C++ are one of the most powerful sequence containers, offering dynamic resizing and memory management capabilities.



Vector Operations

```
vector<int> v;
v.push_back(2);          // [2]
v.push_back(3);          // [2, 3]
v.push_back(7);          // [2, 3, 7]
v.push_back(5);          // [2, 3, 7, 5]
v[1] = 4;                // sets element at index 1 to 4 -> [2, 4, 7, 5]
v.erase(v.begin() + 1);  // removes element at index 1 -> [2, 7, 5]
// this remove method is O(n); to be avoided
v.push_back(8);        // [2, 7, 5, 8]
v.erase(v.end() - 1);  // [2, 7, 5]
// here, we remove the element from the end of the list; this is O(1).
v.push_back(4);                    // [2, 7, 5, 4]
v.push_back(4);                    // [2, 7, 5, 4, 4]
v.push_back(9);                    // [2, 7, 5, 4, 4, 9]
cout << v[2];                      // 5
v.erase(v.begin(), v.begin() + 3);  // [4, 4, 9]
// this erases the first three elements; O(n)
```

NB: Both back() and front() allow direct access to elements and are useful when you need to quickly get or modify the first or last element of a vector.

Sets

Definition

Set Data Structure is a type of data structure which stores a collection of distinct elements.
The C++ standard library contains two set implementations: The structure set is based on a balanced binary tree and its operations work in O(log n) time. The structure unordered_set uses hashing, and its operations work in O(1) time on average. The choice of which set implementation to use is often a matter of taste.
The benefit of the set structure is that it maintains the order of the elements and provides functions that are not available in unordered_set. On the other hand, unordered_set can be more efficient.

Set vs Unordered_set

	set	unordered_set
1.	It is used to store the unique elements.	It is used to store the unique elements.
2.	Sets are implemented using Binary search trees.	It is implemented using hash table
3.	It stores the elements in increasing order.	It stores the element with no order.
4.	We can traverse sets using iterators.	We can traverse unordered_set using iterators.
5.	It is included in #include <set> header file.	It is included in #include <unordered_set> header file.

Set Operations

```
set<int> s;
s.insert(3);
s.insert(2);
s.insert(5);
cout << s.count(3) << "\n"; // 1
cout << s.count(4) << "\n"; // 0
s.erase(3);
s.insert(4);
cout << s.count(3) << "\n"; // 0
cout << s.count(4) << "\n"; // 1
```

```
set<int> s = {2,5,6,8};
cout << s.size() << "\n"; // 4
for (auto x : s) {
    cout << x << "\n";
}
```

```
set<int> s;
s.insert(5);
s.insert(5);
s.insert(5);
cout << s.count(5) << "\n"; // 1
```

Multiset

```
multiset<int> s;
s.insert(5);
s.insert(5);
s.insert(5);
cout << s.count(5) << "\n"; // 3
```

```
s.erase(5);
cout << s.count(5) << "\n"; // 0
```

```
s.erase(s.find(5));
cout << s.count(5) << "\n"; // 2
```

Maps

Definition

A map is a generalized array that consists of key-value-pairs. While the keys in an ordinary array are always the consecutive integers 0,1,...,n 1, where n is the size of the array, the keys in a map can be of any data type and they do not have to be consecutive values. The C++ standard library contains two map implementations that correspond to the set implementations: the structure map is based on a balanced binary tree and accessing elements takes O(log n) time, while the structure unordered_map uses hashing and accessing elements takes O(1) time on average.

Map vs Unordered_map

	map	unordered_map
1.	map is define in #include <map> header file	unordered_map is defined in #include <unordered_map> header file
2.	It is implemented by red-black tree.	It is implemented using hash table.
3.	It is slow.	It is fast.
4.	Time complexity for operations is O(log N)	Time complexity for operations is O(1)
5.	map is used to store elements as key,value pairs in order sorted by key.	unordered_map is used to store elements as key,value pairs in non-sorted order.

Map Operations

```
map<string,int> m;
m["monkey"] = 4;
m["banana"] = 3;
m["harpsichord"] = 9;
cout << m["banana"] << "\n"; // 3
```

```
map<string,int> m;
cout << m["aybabtu"] << "\n"; // 0
```

```
if (m.count("aybabtu")) {
    // key exists
}
```

```
for (auto x : m) {
    cout << x.first << " " << x.second << "\n";
}
```

```
map<string, int> people = { {"John", 32}, {"Adele", 45}, {"Bo", 29} };
cout << "Adele is: " << people.at("Adele") << "\n"; //Adele is: 45
cout << "Bo is: " << people.at("Bo") << "\n"; //Bo is: 29
                // Add new elements
                people.insert({"Jenny", 22});
                people.insert({"Liam", 24});
```

```
// Trying to add two elements with equal keys
people.insert({"Jenny", 22});
people.insert({"Jenny", 30});
cout << "Jenny is: " << people.at("Jenny") << "\n"; //=> Jenny is: 22
```

```
// Remove an element by key
people.erase("John");
```

```
// Remove all elements
people.clear();
```

To find out how many elements a map has, use the `.size()` function

```
cout << people.size(); // Outputs 3
```

The `.empty()` function returns `1` (*true*) if the map is empty and `0` (*false*)

Pairs

Definition

If we want to store a collection of points on the 2D plane, then we can use a dynamic array of pairs.
Both vector<vector<int» and vector<array<int,2» would suffice for this case, but a pair can also store two elements of different types.

Pair Operations

- **pair<type1, type2> p:** Creates a pair p with two elements with the first one being of type1 and the second one being of type2.
- **make_pair(a, b)**: Returns a pair with values a, b.
- **{a, b}**: With C++11 and above, this can be used as to create a pair, which is easier to write than make_pair(a, b).
- **pair.first**: The first value of the pair.
- **pair.second**: The second value of the pair.

Tuples

Definition

We can hold more than two values with something like pair<int, pair<int, int», but it gets messy when you need a lot of elements. In this case, using tuples might be more convenient.

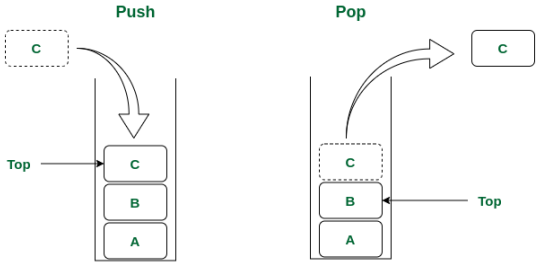
Tuple Operations

- **tuple<type1, type2, ..., typeN> t:** Creates a tuple with N elements, iťh one being of typei.
- **make_tuple(a, b, c, ..., d):** Returns a tuple with values written in the brackets.
- **get<i>(t):** Returns the iťh element of the tuple t. Can also be used to change the element of a tuple.
  This operation only works for constant i. Namely, it is not allowed to do something like the following since i is not constant:
- **tie(a, b, c, ..., d) = t :** Assigns a, b, c, ..., d to the elements of the tuple t accordingly.

Stack

Definition

Stack is a linear data structure which follows a particular order in which the operations are performed. The order may be LIFO(Last In First Out) or FILO(First In Last Out). In stack, all insertion and deletion are permitted at only one end of the list.



Stack Data Structure

Stack Operations:

- **push():** When this operation is performed, an element is inserted into the stack.
- **pop():** When this operation is performed, an element is removed from the top of the stack and is returned.
- **top():** This operation will return the last inserted element that is at the top without removing it.
- **size():** This operation will return the size of the stack i.e. the total number of elements present in the stack.
- **empty():** This operation indicates whether the stack is empty or not.

Queue

Introduction

Like Stack, Queue is a linear structure which follows a particular order in which the operations are performed. The order is First In First Out (FIFO). In the queue, items are inserted at one end and deleted from the other end. A good example of the queue is any queue of consumers for a resource where the consumer that came first is served first. The difference between stacks and queues is in removing. In a stack we remove the item the most recently added; in a queue, we remove the item the least recently added.

Queue Data Structure

Queue Operations:

- **push():** Adds (or stores) an element to the end of the queue..
- **pop():** Removal of elements from the queue.
- **front() or back():** Acquires the data element available at the front node of the queue without deleting it.
- **empty():** This operation indicates whether the queue is empty or not.
- **size():** This operation will return the size of the queue i.e. the total number of elements present in the queue.

Practice

Thank you for listening !

Idriss El Abidi
Souhail Louzi

Special thanks to the former CP leads:
Hamza BA-MOHAMMED, Hachim Hassani Lahsinui and mohamed ghacham amrani
for their insightful review and precious remarks.