Linear Data Structures

BPT Training Session 1

Quick Note

- During IEEEXtreme (and most, if not all other competitions) you are allowed to look up documentation for your programming language!
- Don’t need to memorize all of the information in this presentation
- Key takeaways:
  - **Know what your options are** so you can look them up during the competition
  - Get an idea of the **strengths and weaknesses** of each data structure

Static Array

Arrays are natively supported in C++ and Java. They are very fast, but they have a fixed length which must be specified when they are created.

- Arrays allow *constant time* access to any item (random access)
- If you don't know the *exact* size needed, but you do know the *maximum* size, arrays are still useful -- you don't need to use all of the space you reserve!
- Arrays are ideal for **sorting,** and **searching** data which has been sorted
  - C++: `sort` and `binary_search` in `<algorithm>`
  - Java: `Arrays.sort`, `Arrays.binarySearch` in `java.util.Arrays`

Arrays in C++

Stack Declaration

```
int arr[10];
```

Heap Declaration

```
int* arr = new int[size];
// ...
delete[] arr;
```

Arrays in C++

Initialization

```
int arr[3]{0}; // [0, 0, 0]
int arr[3]{1, 2, 3}; // [1, 2, 3]
int arr[]{1, 2, 3}; // [1, 2, 3]
int arr[2]{1, 2, 3}; // error: too many initializers
int arr[5]{1, 2, 3}; // [1, 2, 3, 0, 0]
int* arr = new int[]{1, 2, 3}; // [1, 2, 3]
```

Optional '='

```
int arr[3] = {1, 2, 3}; // [1, 2, 3]
```

Arrays in Java

- You can use an initializer list
- Otherwise, the array is zero-initialized (`0`, `'\0'`, `false`, etc.)

```java
int[] arr = new int[size];
int[] arr = new int[]{1, 2, 3};
```

Accessing Array Items

- Same syntax for C++ and Java

```
int x = arr[index];
arr[index] = value;
```

Quick Aside: Running Times

In computer science, we aren't usually concerned with *exact* runtimes. **What we care about is how the runtime grows as the size of the problem grows** (such as the number of items in a data structure)**.**

- Constant time: runtime stays the same for any problem size
  - We don't care what the exact length of the constant time is, what matters is that it will be efficient even for huge test cases.
- Linear time: runtime increases proportionally to problem size
  - For example, if we want to sum all elements in an array, we have to check each item once.
  - So, as the array size (problem size) increases, the time to sum the elements increases too.
  - Therefore, the sum operation runs in linear time.

Dynamically-Resizable Array

- Do not have a fixed size
  - Inserting and removing elements after the last item takes *constant time*
  - Inserting or removing elements anywhere else is slow because elements must be shifted over
- Random access (all elements accessed in *constant time*)
- Like static arrays, they are great for **sorting**, and **searching** sorted data
  - C++: `sort` and `binary_search` in `<algorithm>`
  - Java: `Collections.sort`, `Collections.binarySearch` in `java.util.Collections`
- C++: **`#include <vector>`**
- Java: **`import java.util.ArrayList;`**

Linked List

Unlike static and dynamic arrays, items in a linked list are not stored in a contiguous sequence. They can be scattered throughout memory, and each item stores the position of the items before and after it.

- Inserting or deleting elements is fast at any place in the list
  - Inserting or deleting any number of items in a linked list takes roughly the same amount of time as one item in a dynamic array
- Accessing the first and last elements takes *constant time*
- Accessing any other element is much slower (no random access)
- C++: `#include <list>`
- Java: `import java.util.LinkedList;`

Example Time

Let’s see an example of using linked lists to quickly delete many items

Stacks and Queues

Both are similar to lists, but…

- **Stack:** items can only be inserted at the top, and only removed from the top
  - Can only view the item at the top
  - Behind the scenes, the *top* is usually the first item in the “list”. Also called the *head.*
- **Queue:** items can only be inserted at the back, and removed from the front
  - Can only view the item at the front

Pros and cons

- Inserting, accessing and removing items takes *constant time*
  - … but you can only do so at one location!
  - This doesn’t make stacks or queues bad, though. Lists, stacks and queues are all used for different things.

Stacks

- Placing an item on the top of the stack: **push**
- Getting the value of the top item without removing it: **peek**
- Removing the item from the top of the stack: **pop**
- Stacks are referred to as First-in Last-out (FILO) or Last-in First-out (LIFO)

Queues

- Have a **front / head** and **back / tail**
- Adding an item to the back: **enqueue**
- Getting the value of the front item without removing it: **peek**
- Removing the item from the front of the queue: **dequeue**
- Queues are referred to as First-in First-out (FIFO)

Using Stacks and Queues in C++ and Java

Both languages have a data structure called the **deque,** or double-ended queue.

- C++: `#include <deque>`
- Java: `import java.util.ArrayDeque;`

Deques can be used in place of a stack or a queue — they allow insertion, access and removal at both ends (but nowhere else).

- Stack: only use methods to insert, peek and remove items at the front
- Queue: only use methods to insert at the back, and peek and remove items at the front

Why use Deques?

- In C++, the built-in **`stack`** and **`queue`** are *container adaptors*
    - In other words, they're just deques (by default) which only let you do certain operations
    - Other than the simplicity of only remembering one data structure, there's no reason not to use them, though.
- From the documentation from Java's **`ArrayDeque`**:

  This class is likely to be faster than `Stack` when used as a stack, and faster than `LinkedList` when used as a queue.

    - Not only does Java not have a built-in queue, the ArrayDeque beats the stack at its own game

Using Deques as Stacks

<pre>using std::deque;
deque<int> stack;
// push
stack.push_front(1);
// peek
int i = stack.front();
// pop (no return value)
stack.pop_front();</pre>
	<pre>ArrayDeque<Integer> stack =
	    new ArrayDeque<>();
	// push
	stack.push(1);
	// peek
	int i = stack.peek();
	// pop (returns the item)
	i = stack.pop();</pre>

Using Deques as Queues

```cpp
using std::deque;
deque<int> queue;
// enqueue
queue.push_back(1);
// peek
int i = queue.front();
// dequeue (no return value)
queue.pop_front();
```

```java
ArrayDeque<Integer> queue =
    new ArrayDeque<>();
// enqueue
queue.offer(1);
// peek
int i = queue.peek();
// dequeue (returns the item)
i = queue.poll();
```

Let’s Solve a Problem!

Bracket Matching: given a string containing only brackets ‘(‘, ‘)’, ‘[‘, ‘]’, ‘{‘ and ‘}’, output “Correct” if the brackets are valid, or “Incorrect” if they are invalid

- Proper competitive programming problems will be much more specific about what is “valid,” but for this example think about the rules enforced by programming languages.
- Which data structure do we need? Let’s try solving an example ourselves and see if we can find out.

Extra Tip!

- During IEEEXtreme, you are allowed to check the documentation for your programming language. For example, if you forget the name of the method to add an item to a stack, you can look it up!
- C++: https://cplusplus.com/ or https://en.cppreference.com/
- Java: https://docs.oracle.com/javase/8/docs/api/index.html
    - (Java 8 isn't the most recent, but it's a safe bet any competition will use 8 or newer)

Bonus Non-linear Data Structures

- Since this is the only training session before IEEEXtreme, I will quickly introduce you to two non-linear data structures that will probably come in handy.
- We may not have enough time left to cover them in depth, so I encourage you to look them up before (or during) the competition!

Map

- Maps store pairs of items: a *key* and a *value*
- The key and value don't have to be the same type
    - You could map strings to the number of times the letter 'A' appears in each string
- Accessing items: lookup the key, get the value mapped to that key
    - So looking up "apple" would return 1, if we've already created that mapping
    - Logarithmic time (as problem size increases, runtime increases slower and slower)
- C++: `#include <map>` or `<unordered_map>`
- Java: `import java.util.TreeMap` or `.HashMap`
- Unordered / Hash maps will be faster, but map and TreeMap items are automatically sorted (key-value pairs are sorted by their keys)

Priority Queue

- Like a queue, but items are automatically sorted when *enqueued*
  - So whenever we *dequeue* an item, it will always be the item with the min or max value
  - Insertion now takes logarithmic time instead of constant, but it's still relatively fast
  - Dequeuing is still constant time
- C++: `#include <priority_queue>`
  - Items are sorted so that the item you remove has the **maximum** value
- Java: `import java.util.PriorityQueue;`
  - Items are sorted so that the item you remove has the **minimum** value

Thank you for coming!

Code and slides will be posted in the UMIEEE Discord