NOI.PH Training: DS 1

Basic Data Structures

Kevin Charles Atienza and Rene Josiah Quinto

Contents

1 Introduction

A **data structure** is a way of structuring/organizing data in a computer.

For this module, we'll be tackling data structures, both theory and implementation. There are a lot of discussions and implementations of basic and mid-level data structures. If you're already familiar with a lot of the material, then feel free to skim it. This document is pretty comprehensive, so you're bound to find something you may not know about.

1.1 Why data structures?

Why should we care about data structures? Aren't variables enough? Maybe arrays as well?

It shouldn't be hard to motivate why we need to organize data, or even need to organize anything. Consider the following scenario:

> *You are a librarian tasked with maintaining a huge library. You have several shelves for use in storing thousands of books. Specifically, there are two events that might happen at any time:*
>
> - **return**(x). *A person returns a book x he borrowed earlier.*
> - **retrieve**(x). *A person visits the library and wants to borrow book x, if it is in the library. Your task is to find it, retrieve it, and lend it to the person, or determine if it it isn't in the library (or has already been borrowed).*
>
> *How would you do it?*

Now, consider two kinds of people and how they would go about handling this task: the *messy one* and the *organized one*.

- The *messy one* simply puts the book in some shelf in some order. He doesn't care about organizing the books because in his mind, it doesn't matter; after all, the books are in there somewhere.
- The *organized one* groups the shelves into 27 groups. Each group will contain all books whose titles start with a particular letter of the alphabet, with the 27th shelf reserved for all books not starting with a letter. Furthermore, in each group, all books will be sorted alphabetically.

Now, which one do you think will have a better time managing the library? (*Hint:* It is the latter.)

The problem with the messy one's approach is that **retrieve** takes too long. Every time he wants to retrieve a book, he has to go through *the whole library*. Every time. This is terrible; not only is it tiring, it also takes too long, and the person waiting may just leave out of boredom.

In contrast, the organized one can simply go to the appropriate shelf group (corresponding to the first letter of the title), and since the books are sorted alphabetically, he can easily find the target book with something like *binary search*. So, not only will the organized one have a better *time* managing the library, she will also have a better *running time*!

The same will be true if we add a few more kinds of events (or *queries*, or *operations*) such as determining the number of books whose titles begin with a certain letter, or finding

all the books made by some author. Note that the organized one will also find the "finding all books made by some author" hard to perform, but he could easily create some sort of log book containing all books written by every author, so he can easily know where to find all the books and easily retrieve them all.

Similarly, in terms of data structures, having the "arrays are enough" mindset is basically being the messy one. In terms of competitive programming, this will probably mean that you will get a Time Limit Exceeded verdict, since your operations are too slow.

1.2 Concepts and Terms

To make the discussion clearer, we distinguish between two things: **abstract data types** and **data structures**.

- An **abstract data type**, or **ADT**, is a set of values and a set of operations between them, described from the perspective of the *user*, without regard for how it is implemented behind the scenes.
- A **data structure** is the *implementation* or *realization* of an abstract data type. This is the behind-the-scenes of an ADT.

You are probably familiar with data types. When talking about data structures, we usually mean implementing a custom "data type", whose behavior is something we define. The "data type" probably doesn't exist in the language, but could possibly be built from existing ones. In effect, we're *extending* the language to make it easier to do what we want. An ADT is a description of how this new data type should behave, and a data structure is how we implement a data type using already-existing types.

Example 1.1. For example, with the library scenario, the abstract data type is essentially the problem statement—the description that the library acts as a storage, and the behavior of the two operations—because it states the operations that we want to do.

On the other hand, the data structure is how it is implemented behind the scenes, so the messy one and the organized one used different data structures to implement the same abstract data type, one of which performs better than the other.

The previous example illustrates that an ADT can be implemented with multiple data structures. The converse is also true, that is, a data structure can possibly implement multiple ADTs.

Example 1.2. As another example, we can look at *array*s. Actually, we can distinguish between two things called "arrays":

- the **array abstract data type**, which is a fixed-size data storage indexed by nonnegative integers. It supports three kinds of operations:
  - **array.create**(n). Returns an array a with a length of n. The array will be indexed from 0 to $n-1$.
  - a**.get**(i). Return the value at index i.
  - a**.set**(i, v). Set the value at index i to be v.

- a**.length()**. Returns the length of the array.

- the **array data structure**, which is a contiguous set of cells in memory, which implements the array data type. It represents an array a as a pair $a = (f, n)$, where f is the address of the first cell, and n is the length of the array. It implements the operations in the following ways:

  - **array.create(**n**)**. It reserves a contiguous set of n cells from the available memory and returns (f, n), where f is the address of the first allocated cell.
  - a**.get(**i**)**. Returns the value at memory address $a.f + i$. It raises an error if $i < 0$ or $i \geq a.n$.
  - a**.set(**i**,** v**)**. Writes the value v at memory address $a.f + i$. It raises an error if $i < 0$ or $i \geq a.n$.
  - a**.length()**. Returns $a.n$.

Notice that the abstract data type is simply a description of the behavior, and the data structure describes how it is done behind the scenes.

So how come you don't hear of these two kinds of "arrays"? Well, the main reason is that the concept of an *array* is so fundamental and simple that we use the same name for both concepts. Indeed, when we think of an "array", we usually imagine a fixed sequence of blocks, which so happens to be the most straightforward way to implement it in a computer as well.

However, it is very important to know at the back of your mind that the word *array* contains two distinct concepts in it. This will be useful later on as we discover that the *array data structure* can be used to implement many other abstract data types, and the *array abstract data type* can be implemented with many other kinds of data structures. For example, the *dynamic array* ADT can be implemented with arrays, or linked lists, or even trees.

But why should we care if an ADT can be implemented with many kinds of data structures? Why don't we just choose one? Well, the main reason is that every data structure has its own strengths and weaknesses, the most impactful of which is the running time of the operations. We usually want the operations to be performed as quickly as possible,[1] and the constraints are usually set up so that only the (asymptotically) best data structures pass.

This is where it gets interesting and where creativity gets involved; implementing the operations of an ADT *efficiently* can be great problems/puzzles on their own, so much that many problems are explicitly of the "implement the following operations" type.

Harder problems require you to realize that you need a certain ADT *and then* to implement them efficiently (with data structures). But the nice thing about keeping the concepts of "ADT" and "data structure" separate is that it allows you to break down a (possibly complex) problem into two problems, independent of each other: implementing the required ADT (without regard for its use), and solving the problem itself with the ADT (without regard for its implementation). In solving complex problems, this is good—after you've coded the behavior, you don't need to know what happens to the data internally in order to use them.

Exercise 1.3. Describe a data structure that implements the ADT which we'll call the *poppable array*, which is the same as the array ADT with the addition of the following operation:

- a**.pop()**. Remove the last element of the array.

[1] and secondarily with as low memory footprint as possible

The **.create(n)** operation must take $\mathcal{O}(n)$ time, while the rest must take $\mathcal{O}(1)$ time each.

Hint: You can use the *array data structure* for this.

Example 1.4. Even more simply, we can think of an **int** as a data structure that implements the *integer* abstract data type. Integers are an ADT because they are usually thought of as satisfying some mathematical properties and having some arithmetic operations, without regard to how they are implemented behind the scenes.

However, a computer implements an integer with a consecutive sequence of bits (1 or 0) with most of them containing the binary digits of the integer, but one bit is used for the *sign* of the integer.[a] This can be thought of as the data structure supporting the integer data type. But as users of integers, we don't really think of the implementation, only the behavior.

Actually, since C++'s **int**s are bounded in size, strictly speaking it does not represent the integer data type. Instead, it implements the *bounded integer* data type.

[a] More precisely, the bits contain the two's-complement representation of the integer.

Usually, an ADT has a **.create** operation which create an *instance* of that ADT which has a state that is independent of any other instance. We also usually include a **.destroy** operation which destroys that instance, and possibly frees up some memory, but this is usually easy, so we won't discuss it that much.

1.3 C++ Classes

Before we start with data structures, let me describe briefly what a C++ class is, at least with details that are relevant to us. If you wish to go deeper, you can read up on Object Oriented Programming (OOP).

You can think of a **class** as a template for creating objects. An object contains some data representing its state, called *attributes*, and has a certain behavior, defined by its *methods*. Attributes and methods are also called *member variables* or *member functions*, respectively. Thus, **class**es are the perfect tool to use when implementing data structures!

The following code is an example of a class declaration and instantiation. If you're familiar with C++'s **struct**s, then you can think of **class**es like **struct**s, except that, apart from the data members we can usually see in structs, they can also hold functions that are local to them.

```
#include <bits/stdc++.h>
using namespace std;

class CustomClass {
public:
    int integer;
    char str[100];
    double realnumber;

    CustomClass() {}

    CustomClass(int x, char y[], double z) {
        integer = x;
        strcpy(str, y);
        realnumber = z;
```

```
16      }
17
18      ~CustomClass() {}
19
20      void foo() {
21          integer = int(realnumber/3);
22      }
23
24      int bar() {
25          return integer;
26      }
27  };
28
29  int main() {
30      CustomClass instance1;
31      CustomClass instance2(1, "Hello World", -3.54);
32      instance2.foo();
33      cout << instance2.bar() << '\n';
34  }
```

In the example above, we define two ways to create an instance of the class (also known as *instantiation*), and can be seen in lines 10 to 16. These are called *constructor*s, and they look like functions with the return type omitted and the function name being the same as the class name. The `main` function shows how each constructor is used. If there are no parameters included in instantiation, then the blank constructor is called. Otherwise, the appropriate constructor (the one matching the argument types) is called, which simply copies the values and stores it into the instance. You're free to create whatever constructor you want – just know that constructors are usually used to initialize the values of the class instance.

At line 18, what we see is a *destructor*. This is called when the class instance is destroyed, and so is the opposite of a constructor. The syntax is similar, but with a tilde ~ preceding the class name. In this sample, it doesn't do anything special, but later on we'll see destructors that call the destructors of its members. If you don't need to do anything for deallocation, we can even omit the destructor entirely.

In the `main` function, we can see how the instance's functions are called, and the syntax is similar to that of STL Data Structures—this is because these built-in data structures are actually **`class`**es, with their own set of functions and constructors.

The **`public`**: keyword simply declares that all data and function members that appear after this line can be used by any other function, including your `main` function. Without it, the user can't access the data or functions of the instance of this class. [2]

[2]The significance of the visibility of class members becomes more apparent when considering the class as implementing some *abstract data type*. If we are tasked to provide a class that supports certain operations/functions, then we may take extra steps in order to ensure that no one can interfere with the internal workings of the class, intentionally or accidentally. As such, we can choose to only make **`public`** the functions and data members that we allow to be called or modified, and set to **`private`** otherwise. For example, if we are designing a stack, we can publicize the *push* and *pop* operations, but hide the actual data inside to prevent manipulation. Apart from as a security measure, this also simplifies how we interact with the classes by limiting it to only the intended use. However, for our purposes, setting everything as public is fine since we're only looking at small single-file programs.

You can also allocate instances of your class in *heap memory* instead of the *stack memory* with the **new** keyword. For example:

```
int main() {
    CustomClass* instance1 = new CustomClass;
    CustomClass* instance2 = new CustomClass(1, "Hello World", -3.54);
    instance2->foo();
    cout << instance2->bar() << '\n';
    delete instance2;  // deallocates from the heap (and calls the constructor)
}
```

Note the altered syntax when using stuff from the heap.

You generally want to do this if your class is large and you don't want to pass around huge copies of its instances every time you call a function. You also need to do this if your class has a "recursive" definition, such as

```
class Person {
public:
    string name;
    Person* spouse;
};
```

Note that `CustomClass*` is a distinct type from `CustomClass`, and it's called a *pointer type*; it points to something in the heap.

If you don't know about the stack or heap memory yet, or what these pointer things are, a very quick overview is given in Appendix A. We'll also go over them in more detail in a later module.

2 Basic Data Structures

In this section, we'll discuss the very basic data structures. We'll find that some of them are efficient in certain operations, but inefficient in others. The use of the proper data structure that is fit to the problem at hand is key to designing efficient algorithms and solutions.

We will also consider simple abstract data types and how to implement them.

2.1 Arrays

We have already discussed arrays above, in Example 1.2. In short, an array, as a data structure, is a contiguous blocks of memory, indexed from 0 to $n-1$, where n is its declared size. In addition, because of how arrays are contiguous in memory, they are also *cache-friendly*[3]. What this means is that large arrays can perform a lot faster than large non-contiguous data structures (such as linked-lists, which we'll discuss later on) because of the way the operating system handles memory.

The following are some of the properties of arrays. It can store n objects with space complexity $\mathcal{O}(n)$. Given an index p, we can access or modify the element in index p with time complexity $\mathcal{O}(1)$. However, operations such as **delete** and **insert** have time complexity $\mathcal{O}(n)$. The reason for this is obvious – we need to reallocate the whole array! Even if we decide to become slightly smarter and decide not to reallocate, allowing ourselves to have irrelevant data at the end of the array, the worst case is still $\mathcal{O}(n)$. For example, if we have an array with elements $[1, 2, 3, 4, 5]$, and we want to remove 3, then we have to shift 4 and 5 one step to the left. In the worst case, we are asked to remove the very first element of an array of n elements, and so all $n-1$ elements have to be moved one step to the left which gives us a worst-case time complexity of $\mathcal{O}(n)$. Similarly, inserting elements will require us to shift elements one step to the right, and so has the same worst-case time complexity.

It's another story if the array is sorted – for general arrays, getting the largest element is an $\mathcal{O}(n)$ operation, while for sorted arrays it's $\mathcal{O}(1)$. Finding a certain value in the array becomes $\mathcal{O}(\lg n)$ instead of $\mathcal{O}(n)$ through binary search.

In summary, the array is a fast and versatile data structure, capable of supporting several operations efficiently. However, it has its weaknesses such as insertion, deletion, and finding. For such operations, and several others, we can make use of other data structures for efficient handling.

2.2 Linked Lists

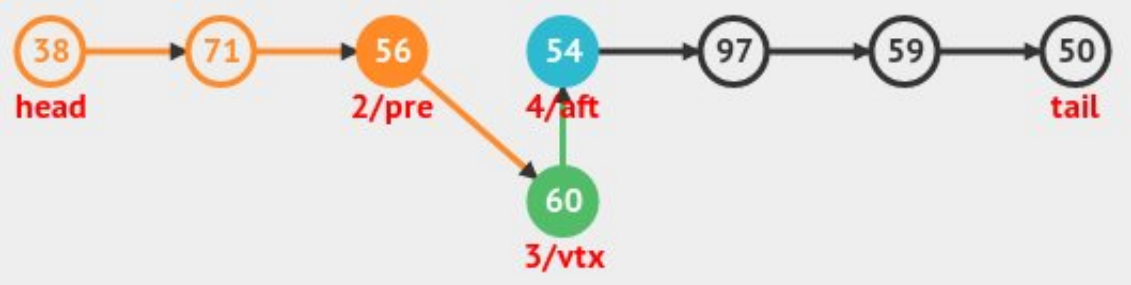
Linked lists (or simply lists), in contrast to arrays, are not stored in contiguous memory locations. Instead, each element is an individual node that contains data, and one element is linked to the next element by a pointer containing the address. The figure below shows a visual representation of a linked list.[4] Note that here, we also show two special nodes - the *head* and the *tail*.

[3] The internal workings of a cache is out of the scope of this document, but it's still an interesting topic. For those who are curious as to how caches work and why they make arrays more efficient, feel free to ask me directly.

[4] You can simulate linked lists visually through VisuAlgo. Most of the images here are also taken from VisuAlgo.

Since linked lists are not stored in contiguous memory, it also means that the data structure is not indexed. In other words, in order to access the p'th element in the linked list, we have to start from the *head* and go to the next node $p-1$ times. The worst case complexity of element access is $\mathcal{O}(n)$, which is significantly slower than an array's access complexity of $\mathcal{O}(1)$.

They are, however, efficient in insertion and deletion operations, assuming that we have pointers to where we want to insert/delete. Take the bottom image as an example. To insert the new node containing "60", we simply have to redirect the pointer at the node containing "56" and make it point to the new node. Then the new node will point to the node containing "54". Thus, insertion in the middle only changes two pointers, and so it has a worst case complexity of $\mathcal{O}(1)$ – significantly faster than the $\mathcal{O}(n)$ insertion operation in arrays. Deletion is similarly efficient with complexity $\mathcal{O}(1)$, assuming we have a pointer to the node to be deleted.



2.2.1 Singly-Linked Lists

A basic implementation of a linked list node class containing `char` values is shown below.

```
class LLNode {
public:
    char c;
    LLNode *next;

    LLNode() {
        next = nullptr;
    }
    LLNode(char _c) {
        c = _c;
        next = nullptr;
    }
    LLNode(char _c, LLNode *_next) {
        c = _c;
        next = _next;
    }
};
```

In the above implementation, we assign `next` to `nullptr` initially, pointing it to the null pointer to indicate that there is no next element.

We can then describe a linked list as a chain of these nodes. Then, if we make sure to keep track of the first element (also known as the *head* of the list), then we can traverse to any node within the list. It would also sound logical to keep track of the last element (also known as the *tail* of the list) if we want to add elements at the end of the list efficiently. An implementation of a linked list data structure using the nodes is shown below.

```
class LinkedList {
public:
    LLNode *head, *tail;

    LinkedList() {
        head = tail = nullptr;
    }
    ~LinkedList() {
        LLNode *temp;
        while (head != nullptr) {
            temp = head;
            head = head->next;
            delete temp;
        }
    }
};
```

The only thing of note is the destructor – if we decide to delete an instance of a linked list, then we need to ensure that each node in the list is deleted as well, since they are all declared within the heap – otherwise, we'll have memory leaks, and in extreme cases it's a quick and easy way to make your PC do a forced reboot.

This is pretty bare, but that's 'cause we haven't implemented any functionality yet. What functionality do we implement? It would depend on the problem at hand, but for linked lists some of the usual operations we need are (a) a constructor to initialize a linked list given an array or range of elements, (b) a function to push an element at the back of the list, (c) a function to insert *before* a given node, (d) a function to remove a certain node, and (e) get the length of the current linked list. All of these are easily implemented, and can be seen in the following code.[5]

```
class LinkedList {
public:
    LLNode *head, *tail;
    int length;

    LinkedList() {
        length = 0;
        head = tail = nullptr;
    }

    LinkedList(vector<char> init) {
```

[5]Note that in the constructor that takes in a `vector<char>`, the whole vector is copied when it is called. If you do not want this behavior, you should declare the constructor as `LinkedList(vector<char>& init)` instead, so only a reference is passed. However, this also means that modifying the `vector` will also modify the original vector passed by the caller; if you want to disallow modifying it in the constructor, declare it as `LinkedList(const vector<char>& init)`.

```
12          length = 0;
13          head = tail = nullptr;
14          for (char v : init) {
15              push_back(v);
16          }
17      }
18
19      ~LinkedList() {
20          LLNode *temp;
21          while (head != nullptr) {
22              temp = head;
23              head = head->next;
24              delete temp;
25          }
26      }
27
28      void push_back(char c) {
29          insert_after(tail, c);
30      }
31
32      void push_front(char c) {
33          insert_after(nullptr, c);
34      }
35
36      void pop_back() {  // !!! this is inefficient in a singly-linked list
37          LLNode *before_tail = head;
38          while (before_tail != nullptr && before_tail->next != tail) {
39              before_tail = before_tail->next;
40          }
41          remove_after(before_tail);
42      }
43
44      void pop_front() {
45          remove_after(nullptr);
46      }
47
48      int size() {
49          return length;
50      }
51
52      // inserts a new node after "node" with value c, or before the head if node == nullptr
53      void insert_after(LLNode *node, char c) {
54          length++;
55
56          LLNode *bago;
57          if (node == nullptr) {  // insert to front
58              bago = head = new LLNode(c, head);
59          } else {
60              bago = node->next = new LLNode(c, node->next);
61          }
62          if (tail == node) tail = bago;
63      }
64
65      // removes the node *after* node, or removes the head if node == nullptr
66      void remove_after(LLNode *node) {
67          length--;
68
69          LLNode *to_delete;
70          if (node == nullptr) {
71              to_delete = head;
72              head = to_delete->next;
73          } else {
```

```
74              to_delete = node->next;
75              node->next = to_delete->next;
76          }
77
78          if (tail == to_delete) tail = node;
79          delete to_delete;
80      }
81  };
```

There are a lot more operations that involve linked lists, but these are enough for a proof of concept.

As you can see, all operations run in $\mathcal{O}(1)$ time, except `pop_back` which runs in $\mathcal{O}(n)$. The latter is an inherent limitation of the singly-linked list; since we only keep track of the next node in the list, we can't point `tail` to the new tail efficiently, as we can't traverse backwards. This is where doubly-linked lists come in.

Exercise 2.1. The implementation of `insert_after` and `remove_after` have special cases for when `node`, `head` or `tail` are equal to `nullptr`. Show how we can simplify the implementation by initializing `head` and `tail` not as `nullptr`, but some dummy "linked list node" whose contained `char` is ignored.

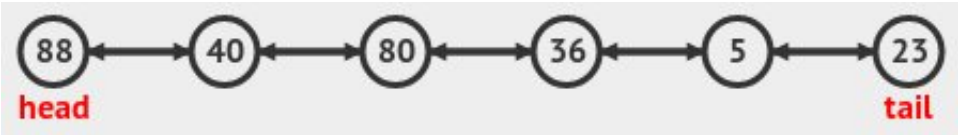
This dummy node, used to signify the end of our structure, is also called a **sentinel node**.

Note: A linked list node can contain something else other than a `char` – indeed, it can contain any data type or multiple data types (even arrays) – just know that the size of your data structure scales directly with the size of your individual node. However, if you somehow need to have multiple linked lists in your code, each containing a different type, then this seems to mean we might have to implement the linked list class multiple times, one for each type. And you know that duplicating code is bad.

What we really need is to have the linked list class be "generic", that is, be able to handle any fixed kind of data type. Luckily, the C++ creators recognize this need and added the feature called *templates* precisely for this. This is why you can have a `vector<int>` and a `vector<char>` in your code; internally, the `vector` class is implemented in the C++ library with a template. Appendix B describes how to use templates to implement generic data structures in general, not just for linked lists.

2.2.2 Doubly-Linked Lists

The previous discussion shows how a basic linked list operates, and also the limitation of only having the next node to keep track of. For doubly-linked lists, we also keep track of the previous node. This means that we can traverse both forward and backward through the list. A representation of a doubly-linked list is shown below.



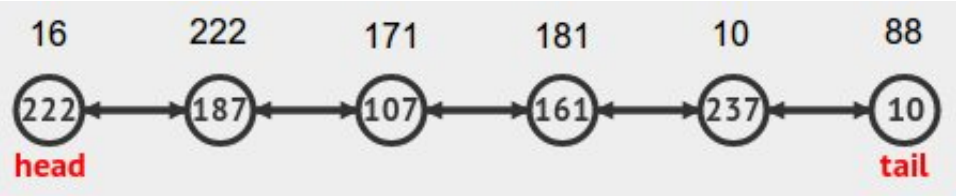
We can implement the same functionality above, and more – because we can traverse backwards, we can now insert and remove elements more efficiently.

```cpp
class LLNode {
public:
    char c;
    LLNode *prev, *next;

    LLNode() {
        prev = next = nullptr;
    }
    LLNode(char _c) {
        c = _c;
        prev = next = nullptr;
    }
    LLNode(char _c, LLNode *_prev, LLNode *_next) {
        c = _c;
        prev = _prev;
        next = _next;
    }
};

class LinkedList {
public:
    LLNode *head, *tail;
    int length;

    LinkedList() {
        length = 0;
        head = tail = nullptr;
    }

    LinkedList(vector<char> init) {
        length = 0;
        head = tail = nullptr;
        for (char v : init) {
            push_back(v);
        }
    }

    ~LinkedList() {
        LLNode *temp;
        while (head != nullptr) {
            temp = head;
            head = head->next;
            delete temp;
        }
    }

    void push_back(char c) {
        insert_after(tail, c);
    }

    void push_front(char c) {
        insert_after(nullptr, c);
    }

    void pop_back() {
        remove(tail);
    }

    void pop_front() {
        remove(head);
    }
```

```
62
63      int size() {
64          return length;
65      }
66
67      // inserts a new node after "node" with value c, or before the head if node == nullptr
68      void insert_after(LLNode *node, char c) {
69          length++;
70
71          LLNode *bago;
72          if (node == nullptr) {  // insert to front
73              bago = head = new LLNode(c, node, head);
74          } else {
75              bago = node->next = new LLNode(c, node, node->next);
76          }
77          if (bago->next != nullptr) bago->next->prev = bago;
78          if (tail == node) tail = bago;
79      }
80
81      void remove(LLNode *node) {  // removes the given node
82          length--;
83
84          if (node->next) node->next->prev = node->prev;
85          if (node->prev) node->prev->next = node->next;
86          if (head == node) head = node->next;
87          if (tail == node) tail = tail->prev;
88
89          delete node;
90      }
91  };
```

2.2.3 XOR Linked Lists

Another variant of linked lists is the **XOR Linked List**, which is a memory optimization of the doubly linked list. Instead of keeping track of two pointers `next` and `prev`, we keep track of the exclusive or ($\oplus$) of the addresses pointed by the two pointers (from this point onwards, we'll refer to this as the *adj-xor* of that node). In other words, it is the xor of values of the pointers. This saves us a single pointer, which might not be much for modern computers, but with enough nodes in the linked lists it can be significant. A figure of an xor linked list is shown below. At the top of the nodes are their addresses in memory, and the numbers inside are the adj-xor values of the nodes.[6]



To traverse the xor linked list, we'll also need to keep track of one of the node's neighboring addresses while moving from node to node – without it, we can't deduce what the neighboring nodes are. For example, while traversing from left to right in the XOR linked list represented above, we can keep track of the next node. If our current address is 222, then the adj-xor stored by this node is 187. The next node address is 171. If we want to move right, we go to the next

[6] Real address values aren't this small. They're either 32-bit or 64-bit, depending on your computer's architecture and your operating system.

node pointed by the address we stored. Once we move, we can calculate the address of the new next node by taking the new node's adj-xor (107) and xor-ing this value to the previous node's address (222). The resulting value (181) is the address of the new next node. From here, if we want to reverse our direction (from traversing from left to right, we go from right to left), we can do so by xor-ing the next node address (181) with the adj-xor (107), and the resulting value is the prev node address (222). If we set this as the next node address for traversal, we're effectively reversing the direction.

We can extend our xor linked list by creating a node, setting the adj-xor of the new node as the address of the tail, xor-ing the adj-xor of the tail to the address of the new node and setting as the tail's new adj-xor, and finally setting the new node as the new tail. This works because the tail's adj-xor value is the prev node's address.

An implementation of the above functionalities is shown below. The `next_hop` and `reverse_dir` functions take in a pair containing the current and next nodes, and it returns a pair containing the new current and next nodes.

```
typedef unsigned long long ull;

typedef class XORLLNode {
public:
    int data;
    ull adj_xor;

    XORLLNode() {}
    XORLLNode(int _data, ull _adj_xor) {
        data = _data;
        adj_xor = _adj_xor;
    }
} XLLN;

class XORLinkedList {
public:
    XORLLNode *head, *tail;

    XORLinkedList() {
        head = tail = nullptr;
    }
    ~XORLinkedList() {
        XORLLNode *temp;
        while (head) {
            temp = head;
            head = (XLLN*)head->adj_xor;
            head->adj_xor ^= (ull)temp;
            delete temp;
        }
    }

    void push_back(int val) {
        if (head == nullptr) {
            head = tail = new XLLN(val, 0LL);
        } else {
            XLLN *new_node = new XLLN(val, (ull)tail);
            tail->adj_xor ^= (ull)new_node;
            tail = new_node;
        }
    }

    pair<XLLN*,ull> next_hop(pair<XLLN*,ull> node) {
```

```
43          XLLN *curr_node = (XLLN*)node.second;
44          XLLN *prev_node = node.first;
45          ull next_node = curr_node->adj_xor ^ (ull)prev_node;
46          return {curr_node, next_node};
47      }
48      pair<XLLN*,ull> reverse_dir(pair<XLLN*,ull> node) {
49          return {node.first, node.second ^ node.first->adj_xor};
50      }
51  };
```

Exercise 2.2. What's interesting about XOR linked lists is that we can reverse sublists in $\mathcal{O}(1)$. More precisely, given two pairs indicating the positions of two end-points of a sublist (and the direction), we can reverse this sublist in $\mathcal{O}(1)$ by modifying at most 4 nodes. Describe how to do this.

If we wanted to do this in doubly-linked lists, we would have to swap the next and prev pointers in each of the nodes in the sublist, having a worst case time complexity of $\mathcal{O}(n)$.

A more "sane" way of implementing these is to store each node as an element of an array.[7] This way, we can use the indices instead of memory addresses to uniquely identify each node[8]. Not only will it simplify coding as we don't have to typecast back and forth from `XLLN*` to `unsigned long long`, but we'll also save more memory – we should be able to store the indices in smaller data types, such as the 32-bit `unsigned int`.[9]

2.3 Stacks and Queues

2.3.1 Theory

The **stack** is a simple abstract data type that can support two operations - **push** and **pop**:

- **stack.create()**. Creates an empty stack, s.
- s**.push(**x**)**. Adds an element to the "top" of the stack.
- s**.pop(**x**)**. Removes the element at the top of the stack and returns it.
- s**.top()**. Returns the top element of the stack.

Stacks follow the Last-In-First-Out (LIFO) principle, similar to, say, a stack of heavy boxes in real life – in order to get an element in the middle of the stack, we have to remove the top elements first.

In some implementations (such as in STL Stack in C++), `pop` does not return the top element; it only removes it.

The **queue** ADT also supports **push** and **pop**, but it behaves differently from stacks.

[7] In fact, we can do the same for linked lists in general, as long as we don't delete nodes often (this leaves gaps in the array, which might be hard to take note of when trying to fill them out).

[8] If we do it this way, you might want to consider reserving index 0 as the "null pointer", and so the elements start at 1.

[9] Actually, you can use this technique of "allocating from an array" for all data structures, and the benefit is usually speed, at the cost of more difficult implementation. I've heard someone call this technique **arena allocation**.

- **queue.create()**. Creates an empty queue, q.
- q**.push(**x**)**. Adds an element to the "back" of the queue.
- q**.pop(**x**)**. Removes the element at the "front" of the queue and returns it.
- q**.front()**. Returns the element at the "front" of the queue.

Queues follow the "First-In-First-Out" principle, similar to real-life queues where the people join the queue at the back, and the front-most person leaves the queue first.

In some implementations (such as in STL Queue in C++), **pop** does not return the front element and only removes it.

Although primitive, the LIFO and FIFO principles manifest themselves in a lot of algorithms, and so queues and stacks are often incorporated in such algorithms.

2.3.2 Implementation

There are several ways to implement stacks and queues. One is to use doubly-linked lists (or even singly-linked lists), as you can implement them so that you can push/pop at the front and at the back efficiently. Such an implementation is shown below. [10]

```
class Stack {
private:
    LinkedList list;

public:
    void push(char c) {
        list.push_back(c);
    }

    void pop() {  // assume non-empty
        list.pop_back();
    }

    int top() {  // assume non-empty
        return list.tail->c;
    }

    int size() {
        return list.size();
    }
};

class Queue {
private:
    LinkedList list;

public:
    void push(char c) {
        list.push_back(c);
    }

    void pop() {  // assume non-empty
        list.pop_front();
```

[10] On another note, we can also see that for the stack and queue implementations, some members are private. This is to provide an example as to how visibility is managed in designing data structures. Again, for our purposes we can simply set everything to public without major repercussions.

```
    }

    int front() {  // assume non-empty
        return list.head->c;
    }

    int size() {
        return list.size();
    }
};
```

A better alternative for stacks and queues is to implement them using arrays. As we've discussed before in the Arrays section, arrays are more internally more efficient in both time and memory, and so it's always preferable to use arrays whenever we can (assuming it doesn't change the algorithm, of course). This will work, however, if we know beforehand the maximum size of the stack/queue. For stacks, the implementation is simply appending to or removing from the end of the array while keeping track of the size. For queues, we can implement what is called a circular queue or "ring buffer". For a quick visualization on how a circular queue works, you can watch this short video (though implementations may vary).

The following is an implementation of stacks and queues using arrays.

```
#define N 1000

class Stack {
private:
    int arr[N];
    int length;

public:
    Stack() {
        length = 0;
    }

    void push(int val) {
        arr[length++] = val;
    }

    void pop() {  // assume non-empty
        length--;
    }

    int top() {  // assume non-empty
        return arr[length-1];
    }

    int size() {
        return length;
    }
};

class Queue {
```

```
private:
    int arr[N];
    int head, rear;

public:
    Queue() {
        head = 0;
        rear = 0;
    }

    void push(int val) {
        arr[rear] = val;
        rear = (rear + 1) % N;
    }

    void pop() {
        head = (head + 1) % N;
    }

    int front() {
        return arr[head];
    }

    int size() {
        return (rear - head + N) % N;
    }
};
```

2.4 Dynamic Arrays

Dynamic arrays are essentially arrays that can be resized. More precisely, it is an ADT that is an extension of the array ADT, which support additional operations that insert or delete from the front or back of the array.

2.4.1 STL Vectors and Deques

In C++, there are two built-in STL data structures that act as dynamic arrays. The first is `vector`, which supports $\mathcal{O}(1)$ insertion[11] at the back of the sequence, effectively increasing the size. The second is `deque`, which is short for **d**ouble-**e**nded **q**u**eu**e, and it supports $\mathcal{O}(1)$ insertion both at the back and the front of the sequence. Dynamic arrays can be used for problems where you aren't given a definite bound for a sequence (such problems are "old-school", and modern/newer problems should not be so vague). You can achieve the same by using functions such as `realloc`, but they add extra complexity that can be avoided with abstraction.

Because of the `push_back` and `pop_back` operations available to vectors and deques, you can use them to implement more versatile versions of stacks. Similarly, because of the `pop_front` operation available to deques, they can be used to implement more versatile versions of stacks and queues.

[11] here, by "$\mathcal{O}(1)$", we mean "amortized $\mathcal{O}(1)$", which is slightly different from "worst-case $\mathcal{O}(1)$". We will discuss this distinction in a future module.

And so the question is – which one is faster? The results of an in-depth experiment to compare vectors and deques can be found in this article. Here are a couple of main points to be taken from the article:

- Deques are more efficient at insertions at the back, and additionally can support insertions at the front.
- Deques are significantly slower at deconstruction, which happens when you use `clear`, and generally when you're done with the deque or replace it with a new deque.
- If unsure which one to use, use vectors.
- If you expect to push several elements to a vector, first call vector's `reserve` function first to indicate how many elements you expect to have e.g. `v.reserve(10000)`. This speeds up vector's average `push_back` time.

In short, if you plan on inserting a lot at either ends of your dynamic array, use deques. Otherwise, if you're just using it for storage or as a container, use vectors. Though these are micro-optimizations, in some cases they can spell out the difference between TLE and AC.

2.4.2 Vector Implementation

It was mentioned above that if we plan on using the `push_back` function of vector a lot (or `insert` for that matter, albeit slow), we should call `reserve` beforehand. But what really happens internally? Internally, vectors still keep the data in arrays (contiguous memory), with two important values to keep track of – `size` and `capacity`. Capacity is how much memory is allocated, and size is how many elements are currently contained inside. This operates in the same way as our Stack implementation using arrays from earlier – `N` is the capacity, and `length` is the size.

So what happens when we try to insert more data, and the size exceeds the capacity? What vectors will do is *reallocate* a new array with double the capacity, and then transfer the contents to this new array. This seems very slow — an $\mathcal{O}(N)$ operation, but remember that since you are doubling the capacity, the next time that the size exceeds the capacity will be much later, and so you only really need to reallocate a few times.[12]

We can avoid having to reallocate in the future using the `reserve` function. What this does is prematurely change the capacity to a large value (it can't be used to decrease capacity). If you set it a high enough value, we won't need to reallocate again at all.

The following is an implementation of a vector data structure.[13] Note that it only supports limited functionality – C++ vectors can do more, such as `insert`, `erase`, and so on (though they are only as efficient as arrays in those).

```
1  class vector {
2  public:
3      int *arr;
4      int curr_size, capacity;
5
6      vector(int n = 1) {
7          arr = new int[n];
```

[12] We will revisit this idea of "only needing to do expensive operations a few times" when we discuss *amortized complexity* later on.

[13] Here, we add a default value for n in the constructor. If n is not provided, then the default value will be passed.

```
        curr_size = 0;
        capacity = n;
    }
    ~vector() {
        delete[] arr;
    }

    void reserve(int n) {
        if (n <= capacity)
            return;

        int *new_arr = new int[n];
        for(int i = 0; i < capacity; i++)
            new_arr[i] = arr[i];

        delete[] arr;
        arr = new_arr;
        capacity = n;
    }

    void push_back(int val) {
        if (curr_size == capacity)
            reserve(2*capacity);

        arr[curr_size++] = val;
    }

    void pop_back() {
        curr_size--;
    }

    int size() {
        return curr_size;
    }
};
```

You can play around with it, for example by creating a vector and pushing 10^8 integers in it, then creating another vector and doing the same but reserving $10^8 + 5$ beforehand. The difference is significant.

Exercise 2.3. With the given `vector` implementation, what is the running time of the following sequence of operations?

- `v.reserve(1)`.
- `v.reserve(2)`.
- `v.reserve(3)`.
- `v.reserve(4)`.
- ...
- `v.reserve(n)`.

Propose an improvement to the implementation of `.reserve` to address this issue.

2.5 Binary Trees

So far, we've been discussing linear data structures. In next few sections, we'll be discussing two basic non-linear data structures. Some of these data structures usually have a certain ordering scheme, which makes them efficient in some operations such as finding, inserting, and deleting values. These first few non-linear data structures we'll tackle all revolve around the concept of a binary tree.

2.5.1 Theory

A **binary tree** is type of rooted tree where each node has at most two children, usually called the *left* and *right* child. In the graphs module, we've talked about a k-ary tree. A binary tree is simply a 2-ary tree. However, a slight difference we'll have here is that we will *distinguish* between the left and the right subtree. Thus, there are actually two distinct binary trees with 2 nodes, not just one.

2.5.2 Implementation

A binary tree node is usually implemented as follows:

```
class BTNode {
public:
    int data;
    BTNode *left, *right;

    BTNode() {
        left = right = nullptr;
    }
    BTNode(int val) {
        data = val;
        left = right = nullptr;
    }
    ~BTNode() {
        if (left) delete left;
        if (right) delete right;
    }
};
```

From the perspective of the code, a `BTNode` is a cousin of the `LLNode`, where instead of a single `next` pointer, there are two "next" pointers.

Similarly to having a `prev` pointer, we can also store the "prev" pointer in a `BTNode`. However, we should call it `parent` since we are in a rooted tree.

```
class BTNode {
public:
    int data;
    BTNode *left, *right, *parent;

    BTNode() {
```

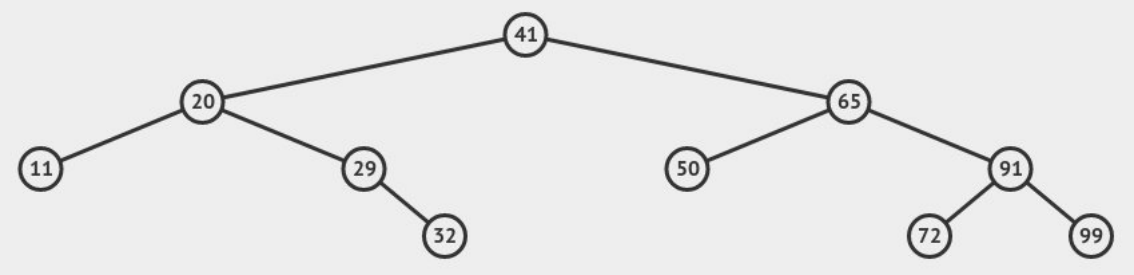
```
7           left = right = parent = nullptr;
8       }
9       BSTNode(int val) {
10          data = val;
11          left = right = parent = nullptr;
12      }
13      ~BTNode() {
14          if (left) delete left;
15          if (right) delete right;
16          if (parent) {
17              // point the parent to null
18              if (parent->left == this)
19                  parent->left = nullptr;
20              else
21                  parent->right = nullptr;
22          }
23      }
24  };
```

This implementation is good for implementing any binary tree, no matter what shape. However, this is not the only way to represent a tree. We shall see later on a way to represent a balanced binary tree compactly with just a single array!

2.6 Binary Search Trees

2.6.1 Theory

A **binary tearch tree** (BST) is a binary tree which has the following ordering scheme: for any node containing a value v, all the elements in the left child's subtree are less than (or equal to) v, and all the elements in the right child's subtree are greater than (or equal to) v. A figure of a BST is shown below.



If we're looking for an element in the tree, we can efficiently search for the value by going down the tree, comparing with the value of the nodes along the way. If the value you're searching for is greater than the value contained in the node, then go to the right child. If it's less, go to the left child. If it's equal, you've found it. In the worst case, we go down as the farthest leaf (a node without children). And so, the worst case complexity reduces to the height of the tree (distance from the root to the farthest leaf). For a balanced tree, this is logarithmic ($\mathcal{O}(\lg n)$). We can also insert in logarithmic time, similar to how we searched, but in this case a node is created and inserted, possibly pushing other nodes down. Deleting a node within a tree is also logarithmic in time, and we take one of its children to be the new subtree root.

These logarithmic complexities presuppose that the tree is well-balanced – if it is not, then the worst case is that the nodes chain up like a linked list, increasing the time complexities to a worst case $\mathcal{O}(n)$. And so, to keep a binary search tree efficient, we have to regularly balance

it. However, we can't completely balance it every time, because converting even a slightly unbalanced tree to a completely balanced one usually takes $\mathcal{O}(n)$. What we need is to keep the tree "balanced enough" that the height is still $\mathcal{O}(\lg n)$, but the amount of work required to balance it after every insertion is also $\mathcal{O}(\lg n)$.

And there are ways to do that. However, tree balancing is out of the scope of this module's topics, and instead will be discussed in the next data structures set. It is also worth mentioning that if elements are inserted in a random order, then the height of the tree is expected to be $\Theta(\lg n)$.

One of the most common uses of the binary search tree is in the implementation of the abstract data type called the **set**, which contains a collection of distinct items, and supports the following operations:

- **set.create()**. Create a new set s.
- s**.insert(**v**)**. Insert the value v to the set (if it doesn't already exist).
- s**.contains(**v**)**. Check whether v is in the set or not.
- s**.remove(**v**)**. Remove the value v from the set.

In fact, common binary search tree implementations implement much more than these operations.

However, note that binary search trees can only contain types where the elements can be *totally ordered* (such as integers or strings).

2.6.2 Implementation

An implementation of a non-balancing binary search tree can be seen below, and implements some of the functionality of the STL Multiset data structure in C++. Note that most of the functions here are recursive, and rightly so – you can consider each node to be its own rooted subtree, and so we can search, insert, and delete from them, and we can even insert/delete entire subtrees.

```
class BST {
private:
    BTNode *root;

    BST() {
        root = nullptr;
    }
    ~BST() {
        if (root)
            delete root;
    }

    BTNode* find(BTNode* root, int val) {
        if (root == nullptr)
            return nullptr;  // not found
        if (val == root->data)
            return root;
        if (val < root->data)
            return find(root->left, val);
        return find(root->right, val);
    }

```

```
23      void insert(BTNode *root, BTNode *bst) {
24          if (bst->data < root->data) {
25              if (root->left == nullptr)
26                  root->left = bst;
27              else
28                  insert(root->left, bst);
29          }
30          else {
31              if (root->right == nullptr)
32                  root->right = bst;
33              else
34                  insert(root->right, bst);
35          }
36      }
37
38      // returns the root of the modified subtree
39      BTNode* _private(BTNode *bst) {
40          if (bst->left) {  // choose left as new root
41              if (bst->right)  // insert the right into the left (combine them)
42                  insert(bst->left, bst->right);
43
44              if (bst->parent) {
45                  if (bst->parent->left == bst)
46                      bst->parent->left = bst->left;
47                  else
48                      bst->parent->right = bst->left;
49              }
50
51              delete bst;
52              return bst->left;
53          } else if (bst->right) {
54              // choose right as new root
55              // left is empty, no need to combine
56
57              if (bst->parent) {
58                  if (bst->parent->left == bst)
59                      bst->parent->left = bst->right;
60                  else
61                      bst->parent->right = bst->right;
62              }
63
64              delete bst;
65              return bst->right;
66          }
67          else {
68              if (bst->parent) {
69                  if (bst->parent->left == bst)
70                      bst->parent->left = nullptr;
71                  else
72                      bst->parent->right = nullptr;
73              }
74              delete bst;
75              return nullptr;
76          }
77      }
78
79  public:
80      // returns a pointer to the node with value val
81      // returns the null pointer if it doesn't exist
82      BTNode* find(int val) {
83          return find(root,val);
84      }
```

```
85
86      // inserts a single value
87      void insert(int val) {
88          BTNode *new_node = new BSTNode(val);
89          insert(root,new_node);
90      }
91
92      // inserts a node (can be a tree itself)
93      void insert(BTNode *bst) {
94          insert(root,bst);
95      }
96
97      // remove a single node
98      void remove(BTNode *node) {
99          BTNode *ret = _private(node);
100         if (node == root)
101             root = ret;
102     }
103
104     //remove a node by value
105     void remove(int val) {
106         remove(find(val));
107     }
108 };
```

Finding and inserting can be easily understood and implemented, but removing has to consider how to connect the parent with its children subtrees (if any). In this implementation, the left child is turned into the root of the subtree, and the right subtree is then reinserted into this new root. The parent is then connected to the new subtree.

Exercise 2.4. Explain how to use binary search trees to implement the ADT called the **dictionary** or **key-value store**, which is a collection of distinct key-value pairs, and where the keys are distinct.

- **dictionary.create()**. Create a new dictionary d.
- d**.set(**k**,** v**)**. Insert the key-value pair (k, v) to the dictionary.
- d**.containsKey(**k**)**. Check whether the key k is in the dictionary or not.
- d**.remove(**k**)**. Remove the key-value pair with key k from the dictionary.

Exercise 2.5. Explain how to use binary search trees to implement the ADT called the **multiset**, which is the same as a set but it can contain the same element multiple times.

2.7 Binary Heaps

2.7.1 Theory

The **binary heap** is also a non-linear data structure, and each node still has at most two children (binary tree). How it differs from a binary search tree is that the ordering of heaps follow is what's called the "heap property" – the root value is greater than (or equal to) the value of its children. In other words, for any sub-tree of the heap, the root of the sub-tree is

the largest element of that sub-tree. This property proves to be useful if we wish to find the largest element fast – it can be found in $\mathcal{O}(1)$. Similar to BSTs, we can also insert and remove elements in $\mathcal{O}(\lg n)$.

The variant described above is called a *max heap*, since the root contains the largest element. You can also consider a *min heap*, and its implementation can be easily inferred from the implementation of the max heap. Thus, for the following discussion, we will only consider max heaps.

One of the main uses of heaps is to implement the abstract data type known as the **priority queue**. A priority queue is an abstract data type[14], with the following property: they are "queues", but instead of following the First-In-First-Out (FIFO) principle, they follow the Highest-Priority-First-Out principle. Specifically, values inserted also have corresponding *priority* values, and **.pop()** and **.top()** operate on the value with the highest priority.

- **priorityQueue.create()**. Create a new priority queue pq.
- pq**.insert(**v**,** p**)**. Insert the value v with priority p.
- pq**.top()**. Return the value with the highest priority.[15]
- pq**.pop()**. Remove the value with the highest priority.

This proves very useful in a lot of algorithms, such as Dijkstra's algorithm for getting the Single Source Shortest Path (SSSP).

Again, these presupposes that the heap is well-balanced, but we'll see later on that this shouldn't be a problem – by the nature of the implementation itself, the heap is always well balanced.

2.7.2 Implementation

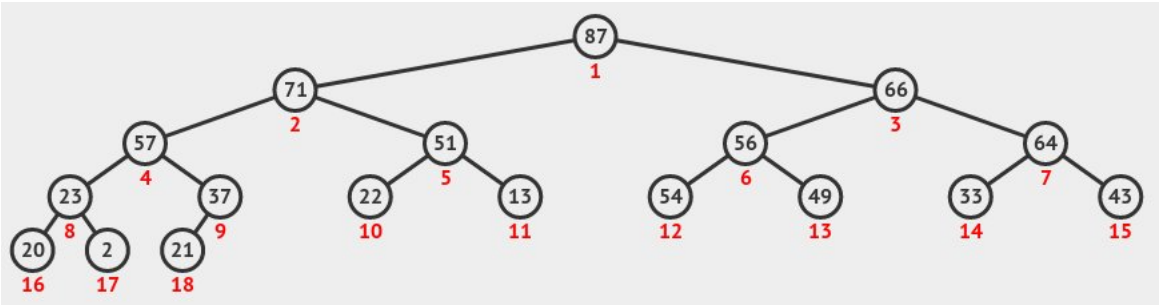
For a good visualization on a lot of the heap functions that we'll be implementing, you can check out this video series.

One of the great advantages of binary heaps over binary search trees is that binary heaps can be implemented using arrays (or a dynamic array, for resizability). Since arrays are both time and memory efficient, this gives a good speedup even if they're doing the same operations. We assign each node an index of the array, and we can do this in two ways:

1. We assign the index 0 to the root node, and for any node with index i, we assign $2i + 1$ and $2i + 2$ to its left and right children, respectively.
2. We assign the index 1 to the root node, and for any node with index i, we assign $2i$ and $2i + 1$ to its left and right children, respectively.

Either one is valid, and choosing one is just a matter of preference. For our implementation, we'll use the second one (root node is 1). The figure below shows what it'll look like. Notice that the heap elements in the figure always fill out the indexes from 1 to n, where n is the number of elements in the heap. This is an important property, and we'll see later on how to maintain and take advantage of it.

[14] You can also implement a priority queue using binary search trees.
[15] If there are multiple values with the same priority, it returns any one.



We have four functionalities that we need to implement - (a) a constructor to initialize the heap using an initial sequence, (b) a function to push elements into the heap, (c) a function to get the value of the root, and (d) a function to remove the root. There can be many more functionalities to consider, but these are the necessary operations for a priority queue, so we'll stick with these.

To insert an element into the heap, what we can do first is simply place this element into the next available space in the array. And since we've discussed before that heap elements always fill out consecutive indexes, we can just append it to the end. Now, by appending it to the end of our elements, we are sure that it is added as a leaf of the heap (unless it's the first element, which would then make it the root). The next step is to preserve the heap property.

If we add just any node as a leaf, we're bound to violate the heap property. There's a simple solution to this – if the inserted node is greater than its parent, we can just swap them. And if that node is greater than its new parent, swap them as well. And so on. This way, we make the node "bubble up" to the proper position, where it is less than (or equal to) its parent, and greater than (or equal to) its children. Since the heap is always balanced, this operation takes $\mathcal{O}(\lg n)$ time. Note that if we swap a node with its parent because that node has a greater value, then by transitivity that node is also greater than (or equal to) its parent's other child (if it exists), and so we can swap away without worry. Another thing to note is that a node's parent is unique, and the parent's index can be easily derived from the current index – if the current node has index $i \neq 0$, then the parent has index $\lfloor \frac{i}{2} \rfloor$.

To remove the root, we can swap the root with the last element (which is a leaf), remove the last element, then "bubble down" the new element by comparing it with the greater child, swapping when necessary, and repeat. This operation also takes $\mathcal{O}(\lg n)$ time.

Finally, in order to construct the heap with n elements, we can simply insert these elements one-by-one with complexity $\mathcal{O}(n \lg n)$. However, we can achieve this in $\mathcal{O}(n)$ by adding the elements to the indices of the array first, then properly order later. To properly order this, we first start with the bottom of the heap (last elements) and with each step move upward to the root (first element). For every node we encounter, we make it bubble down to the proper position, and then this sub-tree becomes a valid heap. Note that we can validly bubble it down because its children are already valid heaps because we start from the bottom to the top. For every node, it swaps at most h times, where h is the height of that sub-tree. Proof of complexity is omitted.[16]

With all that's been discussed, we're ready for the implementation. The following is an implementation of a priority queue using a binary heap. We use a vector instead of an array for resizability.

[16]The gist of the proof is the following. The amount of work done for each node is proportional to its height. There are $\mathcal{O}(2^{H-h})$ nodes of height h, where H is the height of the tree. Hence, the total amount of work is $\mathcal{O}(\sum_{h \geq 0} h 2^{H-h}) = \mathcal{O}(2^H \sum_h \frac{h}{2^h}) = \mathcal{O}(n)$, since $2^H = \Theta(n)$, and the latter infinite sum converges.

```cpp
class PriorityQueue {
private:
    vector<int> arr;

public:
    PriorityQueue() {}
    PriorityQueue(vector<int> &v) {
        arr.resize(1);  // ignore index 0
        arr.insert(arr.end(), v.begin(), v.end());

        for(int i = arr.size() - 1; i >= 1; i--) {
            bubble_down(i);
        }
    }

    void bubble_down(int i) {
        while (2*i < arr.size()) {  // while there is still a child
            int c1 = 2*i, c2 = 2*i+1;
            int q = (c2 >= arr.size() || arr[c1] > arr[c2]) ? c1 : c2;

            if (arr[p] < arr[q]) {
                swap(arr[p], arr[q]);
                p = q;
            } else
                break;
        }
    }

    void bubble_up(int i) {
        while (i > 1) {  // while there is still a parent
            int q = p / 2;
            if (arr[p] > arr[q]) {
                swap(arr[p], arr[q]);
                p = q;
            } else
                break;
        }
    }

    ~PriorityQueue() {}

    int size() {
        return arr.size() - 1; // remember to ignore index 0
    }

    int top() {  // assume that the heap is not empty
        return arr[1];
    }

    void push(int val) {
        arr.push_back(val);
        bubble_up(arr.size() - 1);
    }

    void pop() {  // assume that the heap is not empty
        arr[1] = arr.back();
        arr.pop_back();
        bubble_down(1);
    }
};
```

2.8 Summary

What we've shown so far is the theory behind the basic data structures that we use in competitive programming, as well as basic implementations. I'd like to point out that it is *not* recommended that you implement your own data structures if the operations that you need are supported by the standard STL data structures in C++. You only need to implement custom data structures if they don't exist as built-in data structures or if the built-in data structures don't support what you need.

3 Grids

Grid-type problems show up so often in competitive programming contests that it deserves its own section. Here, we'll learn the best approaches to dealing with these. A good starting point on grids is this link.

3.1 N-Dimensional Grids

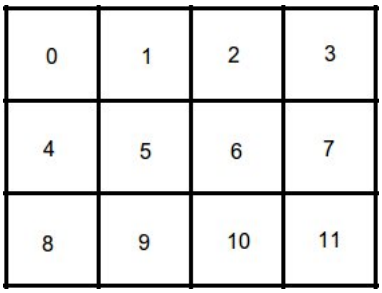
3.1.1 2-Dimensional Grid Representation

The standard grid type is the 2D grids. Whether it's a dynamic programming problem, a graph problem, or even an ad-hoc problem, there's usually at least one per contest. As such, it's important to find efficient ways to handle these problems, coding-wise.

One thing to deal with is representation. It's intuitive enough to represent 2D grids using 2D arrays, as it's the simplest way to represent it. For most problems, this shouldn't present any complications. However, there are some cases where the cap for each dimension is vague. For example, the input can be a 2D grid with dimensions $n \times m$, but you're not given an explicit constraint for n or m. Instead, you're given the constraint $nm \leq 10^6$, which implies that n and m can be as high as 10^6, but the number of elements is at most 10^6. For these types of problems, we can't just create a $10^6 \times 10^6$ array – we don't have enough memory for that (around 931 Gigabytes are needed for 1-byte elements).

For this, we have two options. The first one is to create a vector of vectors, and resize them according to the test case's bounds. Though this ensures that the total *size* does not bloat up to an unmanagable degree, it's harder to manage the total *capacity*. We can use STL vector's `shrink_to_fit` function, which decreases the capacity to exactly the size. This poses problems with overhead – in the worst case, we have to reallocate for every test case. We might as well just create another vector of vectors per test case instead (put the declaration inside the loop).

The second option is to flatten the entire grid into just a single dimension, and then we adjust our indexing. We could number them starting from the top row to the bottom row, and for each row number from left to right. The figure below shows such an indexing scheme.



And so we need two functions – one to map from the 2D index to the new 1D index, and another that maps it backwards. With the indexing scheme above, the functions are easy to derive. For an n-rows by m-columns matrix, the new index for (i, j) is $im + j$. For a flattened array index p, the 2D equivalent is $(\lfloor \frac{p}{m} \rfloor, p \bmod m)$.

Memory-wise and time-wise, this is more efficient (by a constant factor) than the vector approach, as you only need to allocate once at the beginning.

3.1.2 2-Dimensional Grid Traversal

In 2D graph problems, you need to be able to traverse to the adjacent nodes. For some problems these are the nodes in your 4 cardinal directions (North, East, South, and West), and for some it's 8 directions (with the addition of North-East, North-West, South-East, and South-West).

When listing the adjacent nodes, the most straightforward way is to simply list the nodes in all directions. For example, with 4 directions the adjacent nodes to (i, j) are the nodes $(i+1, j)$, $(i, j+1)$, $(i-1, j)$, and $(i-1, j-1)$. However, coding it like this is tedious, especially when you have to type it several times, and even more so if there are 8 directions. Not only is it tedious, but it's also prone to errors. As a general rule: the more code you type out, the greater the chances that there's a typo you missed out. Imagine the nightmare when you're trying to code the adjacent nodes for a knight in a chess board and mistyping a single character (e.g. $(i+1, j+1)$ instead of $(i+1, j+2)$).

The better alternative is to store the directions in what I usually call the *dx-dy* lists. For the 4 cardinal directions, I can store it in this way:

```
constexpr int DIR = 4;
int dx[DIR] = {1,0,-1,0};
int dy[DIR] = {0,1,0,-1};

void list_adjacent(int i, int j) {
    for(int p = 0; p < DIR; p++)
        cout << i+dx[p] << " " << j+dy[p] << endl;
}
```

If we code it this way, we can easily modify the directions with no/minimal changes to the rest of the code. To modify it to support the 8 directions, we only need to change the first three lines:

```
constexpr int DIR = 8;
int dx[DIR] = {1, 1, 1, 0, 0,-1,-1,-1};
int dy[DIR] = {1, 0,-1, 1,-1, 1, 0,-1};
```

We could even support special directions, such as the moves of a knight in chess:

```
constexpr int DIR = 8;
int dx[DIR] = {1, 1,-1,-1, 2, 2,-2,-2};
int dy[DIR] = {2,-2, 2,-2, 1,-1, 1,-1};
```

Even if we do have a typo, it's easier to debug since it'll only be localized to those three lines.

If you really want to ensure that there's no typo even here, you can even choose to programmatically construct the *dx-dy* lists, like so:

```
// 4 cardinal directions
vector<pair<int,int>> dxdy;
```

```
void construct_dx_dy() {
    for (int dx = -1; dx <= 1; dx++) {
        for (int dy = -1; dy <= 1; dy++) {
            if (abs(dx) + abs(dy) == 1) {
                dxdy.push_back({dx, dy});
            }
        }
    }
}
```

```
// 8 cardinal directions
void construct_dx_dy() {
    for (int dx = -1; dx <= 1; dx++) {
        for (int dy = -1; dy <= 1; dy++) {
            if (dx || dy) {
                dxdy.push_back({dx, dy});
            }
        }
    }
}
```

Exercise 3.1. Describe how to programmatically construct all 8 knight moves.

3.1.3 General N-Dimensional Grids

For general N-dimensional grid problems, you are given the number of dimensions N and a list D of length N, each of which is the size of a dimension in the grid. We can flatten it down to a one-dimensional array. We then need two functions for converting indices. To generalize, we can first consider a 3-dimensional grid with dimensions $A \times B \times C$. The mapping from 3D index (i, j, k) to 1D is $iBC + jC + k$. From this, we can infer that the general mapping from ND index $(p_0, p_1, \ldots, p_{N-1})$ to 1D is $p_0 D_1 D_2 \ldots D_{N-1} + p_1 D_2 \ldots D_{N-1} + \ldots + p_{N-1}$. This can be described compactly as:[17]

$$f(p) = \sum_{i=0}^{N-1} \left[p_i \prod_{j=i+1}^{N-1} D_j \right]$$

Though it seems like computing it is an $\mathcal{O}(N^2)$ operation, we can compute it in $\mathcal{O}(N)$ by transforming it into the following recurrence and applying DP.

$$\begin{aligned} g(p, 0) &= p_0 \\ g(p, k) &= g(p, k-1) D_k + p_k \\ f(p) &= g(p, N-1) \end{aligned}$$

From this, we can derive the inverse mapping by working backwards.

[17] For those unfamiliar with the $\prod$ (capital pi) notation, it stands for "product" in the same way that the $\sum$ (capital sigma) notation stands for "sum".

$$
\begin{aligned}
g(p, N-1) &= f(p) \\
g(p, k-1) &= \left\lfloor \frac{g(p,k)}{D_k} \right\rfloor \\
p_k &\equiv g(p,k) \mod D_k
\end{aligned}
$$

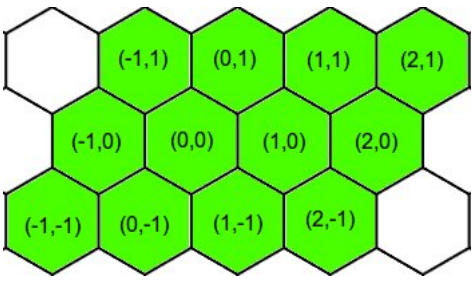
A sample implementation of the two functions is shown below, processed iteratively.[18]

```cpp
vector<int> D;  // size of the dimensions, size is equal to N

int nd_to_flat(vector<int> p) {
    int ret = p[0];
    for(int i = 1; i < p.size(); i++)
        ret = ret*D[i] + p[i];
    return ret;
}

vector<int> flat_to_nd(int x) {
    vector<int> ret(D.size());

    for(int i = D.size()-1; i >= 0; i--) {
        ret[i] = x % D[i];
        x /= D[i];
    }
}
```

As for traversal, we can use the *dx-dy* method, but instead of two arrays, we have N arrays, or in other words it's an $N \times M$ direction table, where M is the number of directions supported.

3.2 Hexagonal Grids

In some problems we may encounter, the grid is *hexagonal* instead of the usual square cells. An example of this is shown in the figure below. This seems intimidating at first, but we can solve problems like these by treating them as 2D grids and using the techniques on 2D grids. To do this, we can imagine that our grid is a parallelogram instead of a rectangle. If we do this, and still keep the 2D coordinates, then we have the coordinates for our hexagonal grid.



[18] This is actually how multidimensional arrays work in C++. Internally, it's a single contiguous array, and the `[]` operators translate the indices to the proper memory location, very similar to what we've shown.

The only problem now is to find a way to get the adjacent nodes. We can create a dx-dy table using the differences in coordinates of the adjacent nodes, which we can see above centering at $(0, 0)$. The implementation of the dx-dy table is also shown below.

```
constexpr int DIR = 6;
int dx[] = { 0, 0,-1,-1, 1, 1};
int dy[] = {-1, 1, 0, 1, 0,-1};
```

You can even construct it programmatically:

```
// 4 cardinal directions
vector<pair<int,int>> dxdy;
void construct_dx_dy() {
    for (int dx = -1; dx <= 1; dx++) {
        for (int dy = -1; dy <= 1; dy++) {
            if (max(abs(dx + dy), max(abs(dx), abs(dy))) == 1) {
                dxdy.push_back({dx, dy});
            }
        }
    }
}
```

3.3 Other Tips and Tricks

When you're traversing the grid, you usually have to constantly check if you've gone over the boundary (something like `if (x<0 || x>=N || y<0 || y>=M)`). This can also be tedious to code, and also prone to errors. One thing we could do is to create a function, say, `in_bounds(x,y)`, which does this check. Another thing we could do is to pad the borders with special values to mark them as *impassable* cells (something like `if (arr[i][j] == -1)` or some special value). This is less prone to errors, as we only need to check if our padding is correct.

4 Data Structures for Range Operations

We will now encounter our first data structures that are *not* built-in data structures in C++. The structures described in this section all attempt to implement the ADT which we'll call the **range-queryable array**, which is an array with the following operations:

- **.create**(a). Create a range-queryable array r with initial values the same as array a.
- r**.inc**(i, v). Increase the value at index i by v. (Traditionally called **point updates**)
- r**.sum**(i, j). Get the sum of all values from index i to index j. (Traditionally called **range sum queries**)
- r**.min**(i, j). Get the sum of all values from index i to index j. (Traditionally called **range minimum queries**)

This is a fairly generic data structure which you can easily imagine to be useful in many applications. It is also very common in programming contests, so much so that structures implementing it is considered basic, sometimes even more basic than self-balancing binary search trees.

Range maximum queries are also easily implemented, based on analogy from **min**, so we will leave it to you to implement.

Some of the data structures here will only implement a subset of the operations mentioned above (and so are not "full" implementations of range-queryable arrays), while some others implement even more operations, with one of them in particular (segment trees) being super flexible that we will revisit it in a later module.

For reference, the most straightforward implementation of this ADT simply is just an *array data structure* with the operations implemented directly. This gives $\mathcal{O}(n)$ time for **create**, $\mathcal{O}(1)$ time for **inc** and $\mathcal{O}(n)$ worst-case time for **sum** and **min**, the last of which is too slow for most purposes.[19]

4.1 Prefix Sums

If there are only **sum** operations, then we can perform each of them in $\mathcal{O}(1)$ time (with an $\mathcal{O}(n)$ **create**) by simply computing the prefix sums of the array.[20] The **prefix sum array** of an array a is an array s of size a.length() $+ 1$ where $s[i]$ is the sum $a[0] + a[1] + \ldots + a[i-1]$. Using this array, it is now easy to compute **sum**(i, j); it is simply $s[j+1] - s[i]$.

Exercise 4.1. Explain why $s[j+1] - s[i] = a[i] + a[i+1] + \ldots + a[j]$.

Exercise 4.2. Almost everywhere else, you will see the prefix sum array defined as an array with size a.length(). Our implementation has one more element, beginning with $s[0] = 0$. Explain why this is a superior approach in the context of computing range sums.

[19] However, note that it may happen that an $\mathcal{O}(n)$ worst-case time is enough, for example, when the usage of the structure is not the bottleneck of your solution. In that case, it is actually preferable to use the straightforward implementation to minimize the possibility of bugs. *Premature optimization is the root of all evil.*

[20] This technique is also discussed in the NOI.PH tutorial page: http://noi.ph/tutorial

The prefix sum array itself can be computed with the following recurrence:

$$s[0] = 0$$
$$s[i+1] = s[i] + a[i].$$

Therefore, **create** runs in $\mathcal{O}(n)$ time, and each **sum** runs in $\mathcal{O}(1)$ time, which is possibly the best you could hope for! Obviously, the downside is that modifications (**inc**) are not supported. A more subtle downside is that by computing the range sum as $s[j+1] - s[i]$, we're implicitly assuming that the "addition" is cancelled by "subtraction". This means that operations like **min** cannot be supported. Other data structures are more flexible and can handle "addition" operations more general than integer addition, without requiring the existence of the inverse "subtraction" operation.

It can be extended to handle the case where all **inc** operations come before all **sum** operations.

Exercise 4.3. Explain how to extend this technique to also handle **inc** operations in $\mathcal{O}(1)$ time, assuming that all **inc** operations happen before all **sum** operations.

Exercise 4.4. Explain how to support general **inc** operations in $\mathcal{O}(n)$ time.

4.2 Block Decomposition

4.2.1 Theory

Block decomposition (sometimes called square root decomposition, or sqrt decomposition) is a technique of dividing a range of size n into $\sqrt{n}$ bins, each bin containing $\sqrt{n}$ elements. With this setup, we can implement a lot of functionality and decrease the time complexity by a factor of $\sqrt{n}$. This is done by preprocessing relevant information for each bin.

To explain how it works, let's look at how it can be applied to a specific application: Sqrt decomposition can be used to implement the operations **min**, **sum**, **inc** in $\mathcal{O}(\sqrt{n})$ query/update time with $\mathcal{O}(n)$ time for **create**.[21]

The first step is to decompose the range into around $\sqrt{n}$ bins. We can do this by getting $m = \lfloor\sqrt{n}\rfloor$, and assigning each element with index i to bin $\lfloor\frac{i}{m}\rfloor$. This will result in having around $\sqrt{n}$ bins. Next is that for every bin, we find the minimum element and store it. We can do this naively in $\mathcal{O}(n)$ time. After this, we're done with the preprocessing.

Each query will ask for the minimum element within a certain range. It can be easily shown that for any range, at most two bins will be partially inside the range (the two ends of the query), and the rest of the bins are either completely outside of the range or completely inside of the range. The bins that are completely outside can be ignored. For the bins that are completely inside the range, we can get the minimum of all their preprocessed values, and store this value (say x). For all the bins that are partially covered, we can do a naive loop over the range that is covered, get the minimum, and replace x if the new minimum is less than x. This x is our final answer to the query.

[21] This may not seem very impressive considering that segment trees can get it in $\mathcal{O}(\lg n)$ time, as we will learn later. This is simply a proof of concept, and it's not always the case that sqrt decomposition is inferior. In fact, sqrt decomposition can be used in a lot of problems, even those that can't be solved using segment trees.

Since there are around $\sqrt{n}$ bins, the initial value of x can be taken in $\mathcal{O}(\sqrt{n})$. There are at most two partially covered bins, and each bin has at most $\sqrt{n}$ elements, and so the second part also takes $\mathcal{O}(2\sqrt{n})$ time. Everything considered, each query can then be solved in $\mathcal{O}(\sqrt{n})$ time.

For updates, we assign the new value to the given index and preprocess the entire bin it belongs to. This can be done in $\mathcal{O}(\sqrt{n})$ time.

Exercise 4.5. Explain how to use sqrt decomposition to also compute **sum** in $\mathcal{O}(\sqrt{n})$ time (maintaining the other operations' time complexities).

Note that although $\mathcal{O}(\sqrt{n})$ is worse than the $\mathcal{O}(1)$ update of the direct technique and the $\mathcal{O}(1)$ query time of the prefix sum technique, overall it still counts as an improvement because the worst-case time *per operation* is improved. For example, suppose there are q operations. Then the worst-case for the previous techniques is $\mathcal{O}(qn)$, while the worst-case for sqrt decomposition is $\mathcal{O}(q\sqrt{n}+n)$. If $q = n = 80000$, then the former will TLE, while the latter will pass.

4.2.2 Implementation

An implementation of the above algorithm is shown below.[22]

```
class SqrtDecomp {
public:
    vector<int> arr;
    vector<int> block_min;
    int m, n;

    SqrtDecomp() {}

    SqrtDecomp(const vector<int>& init) {
        arr = init;
        n = arr.size();
        m = int(sqrt(n));
        for (int i = 0; i < n; i++) {
            int bin = i / m;
            if (bin >= block_min.size()) block_min.push_back(INF);
            block_min[bin] = min(block_min[bin], arr[i]);
        }
    }

    void set(int p, int x) {
        arr[p] = x;
        int bin = p / m;
        block_min[bin] = INF;
        for(int i = bin*m; i < (bin+1)*m && i < n; i++)
            block_min[bin] = min(block_min[bin], arr[i]);
    }

    void inc(int p, int v) {
        set(p, arr[p] + v);
    }

    int range_min(int a, int b) {

        int x = INF;
```

[22] Note that this is only one application of sqrt decomposition. We can implement it to several other range-based query problems.

```
35
36          // go through the partially contained blocks
37          while (a % m) x = min(x, arr[a++]);
38          b++;
39          while (b % m) x = min(x, arr[--b]);
40
41          // go through the completely contained blocks
42          for (a /= m, b /= m; a < b; a++) x = min(x, block_min[a]);
43
44          return x;
45      }
46  };
```

Now, why do we split it to around $\sqrt{n}$ bins? Let's look at the case if we set m to a fixed value (possibly not $\sqrt{m}$). Then the number of bins would be n/m. Now, let's consider the complexity of the query function in terms of these – in the worst case, we'll go through all $n/m - 2$ bins once, and for two end-points we'll go through $m - 1$ elements. This suggests a worst case time complexity of $\mathcal{O}(n/m + m)$. We can try to minimize this total complexity by optimizing the parameter m, and we'll find that $m = \sqrt{n}$ is the optimal value.

We have to note, however, that this chosen m might not be the best when it comes to our implementation. It could be the case that the two "halves" of our query are not symmetric. As such, choosing the proper m can become somewhat of an experiment – we can try $m = 1.5\sqrt{n}$, $m = 0.8\sqrt{n}$, and so on – with the goal in mind that we try to balance the run times of the two halves.[23]

Another approach is to either fix m as a constant, or the number of bins as a constant throughout your program. This way, the program can be sped up through compiler optimization. If we're doing so, we might even want to set our constant as a power of 2, e.g. for a maximum n of 10^5, we can set our constant to be 256 or 512. This way, division, multiplication, and modulo become bit shifts and bit masks, which are a lot faster.

4.3 Segment Tree

4.3.1 What does it do?

A **segment tree** can provide a lot of functionality that operates on the individual elements and ranges, and other features as long as they work under certain properties (more on this later). Most of the functionality it implements can be queried in $\mathcal{O}(\lg n)$, and so they can be very useful when the elements are dynamic.

4.3.2 Theory

We can improve our sqrt decomposition structure even further by subdividing the blocks themselves into even smaller subblocks. This gives our structure three *levels*: the array itself, the list of subblocks, and the list of blocks. When computing the minimum of a block, one only needs to iterate through the subblocks comprising it, not the underlying array itself, making it more efficient. A single update now requires updating a subblock and block, but that's okay.

With a suitable choice of sizes of blocks and subblocks, we can implement all operations in $\mathcal{O}(n^{1/3})$ time each.

[23] If you want to set up an experiment, you can use *ternary search* on your code to find the optimal parameter. You can use the time command in a linux bash tarminal to return a program's runtime.

In fact, nothing is stopping us from adding even more layers, giving us ever-faster running times ($\mathcal{O}(n^{1/4})$, $\mathcal{O}(n^{1/5})$, etc.) as we add more layers! But only up to a certain point; after a while, the number of layers will start to overwhelm and make the update run slower than the range query.

In fact, there's a systematic way to implement this layers-upon-layers idea; it is called a **segment tree**. It works by dividing every block in a layer into two blocks of roughly equal size in the layer below.

A **segment tree** is a flexible data structure that can solve query problems that can be solved using divide-and-conquer on ranges. It works by segmenting the entire range into a hierarchy of ranges, and assigning a node to each range. Leaf nodes cover single elements, while non-leaf nodes cover the combined ranges of its two children (binary tree). Then, each node stores relevant information about its range. For range minimum queries, it can store the minimum element in its range, for range sum queries, it can store the sum of all elements in its range, and so on. As long as we can derive the value of a non-leaf from the value of its children, then segment trees can be a viable solution.

There are generally three types of things to be implemented in a segment tree. The first one is the construction, which takes in an array of values and constructs a balanced binary tree out of them. It does so by repeatedly dividing the array into two halves, constructing both smaller segment trees recursively, and then combining the values at their roots to get the value at the overall root. The base case is on an array of size 1, in which case, a single leaf is created.

The second is an update function, where we update a single index. We go down from the root and visit nodes with ranges that cover our index, until eventually we hit a leaf node. We update the leaf node, and go back up, updating the value of each node along the way.

The last is a query function, which queries a range. Again, we go down from the root and visit nodes with ranges that intersect with our query range. If the node's range is completely within our query range, then we can stop and return the preprocessed value in the node. If the node's range only intersects with our query range, we go down further. For each node, we calculate the answer to the query based on the answers of the children.

The construction has a complexity of $\mathcal{O}(n)$, and update and query functions have a complexity of $\mathcal{O}(\lg n)$. Note that segment trees can support multiple functionalities in the same data structure, e.g., it can support both RMQ and RSQ queries. This is a big improvement over the sqrt decomposition technique!

Later on, we will discover that with additional techniques, we can also implement **range updates** using segment trees in $\mathcal{O}(\lg n)$ time!

4.3.3 Implementation

The following is an implementation of a segment tree. It initially takes in a vector filled with the initial values of the range.

```
const int INF = 1'001'010'101;

class SegmentTree {
private:
    int i, j; // range represented by this node
    int m; // minimum at this range
    SegmentTree *left, *right;

```

```
9       SegmentTree(const vector<int>& init): SegmentTree(init, 0, init.size() - 1) {} // call the
        ↪  other constructor
10
11      SegmentTree(const vector<int>& init, int i, int j): i(i), j(j) {  // initialize i and j
12          if (i == j) {
13              m = init[i];
14              left = right = nullptr;
15          } else {
16              int k = (i + j) / 2;
17              left = new SegmentTree(init, i, k);
18              right = new SegmentTree(init, k + 1, j);
19              combine();
20          }
21      }
22
23      void combine() {
24          m = std::min(left->m, right->m);
25      }
26
27      void inc(int I, int v) {
28          if (i <= I && I <= j) {
29              if (left) {
30                  left->inc(I, v);
31                  right->inc(I, v);
32                  combine();
33              } else {  // leaf node
34                  m += v;
35              }
36          }
37      }
38
39      int min(int I, int J) {
40          if (I <= i && j <= J) {  // completely contained
41              return m;
42          } else if (J < i || j < I) {  // completely disjoint
43              return INF;
44          } else {  // partially overlaps
45              return std::min(left->min(I, J), right->min(I, J));
46          }
47      }
48  };
```

Note that line 9 is just the C++ syntax for a constructor that also calls another constructor. Additionally, line 11 has a part like `: i(i), j(j)`. This is just a convenient way to initialize member functions from arguments passed to the constructor.

Exercise 4.6. Explain why this implementation of **min** runs in $\mathcal{O}(\lg n)$ time.

Hint: At every "level" of the tree, how many subranges *partially* overlap with the query range $[I, J]$?

Exercise 4.7. Explain how to extend this to also handle **sum** (range sum query).

4.4 Binary-Indexed Tree

4.4.1 What does it do?

A **Binary-Indexed tree**[24], or BIT, is another data structure that can support **inc** and **sum**, both in $\mathcal{O}(\lg n)$ time.

Now, you might ask why we should care about binary-indexed trees if segment trees can also implement the same operations with the same time complexity. The reason is that binary-indexed trees use up exactly n cells of memory, as opposed to the $5n$ (or so) required for segment trees, so it uses up less memory. The n cells required are also contiguous in memory, which makes it cache-friendly. On top of that, the code for the operations are so short and simple, that in practice binary-indexed trees run noticeably faster. Although it is just a constant improvement theoretically, it can sometimes convert a TLE code into an AC code.

4.4.2 Theory

Another way to look at *segment trees* is at follows. A segment tree tries to find a set of *special intervals* with the following properties:

1. Every position i is contained in at most $\lg n$ special intervals.
2. Every interval $[i, j] \subset [1, n]$ can be decomposed as a disjoint union of at most $\lg n$ of these special intervals.

The segment tree operations can now be summarized as follows:

- To do a point update at position i, simply iterate through all special intervals containing i and update them all. ($\mathcal{O}(\lg n)$ time)
- To do a range query on $[i, j]$, decompose $[i, j]$ into a disjoined union of special intervals and combine their results. ($\mathcal{O}(\lg n)$ time)

The main idea behind *binary indexed trees* is similar, but the second condition is more relaxed. A BIT tries to find a set of special intervals with the following properties:

1. Every position i is contained in at most $\lg n$ special intervals.
2. Every *prefix* $[1, j] \subset [1, n]$ can be decomposed as a disjoint union of at most $\lg n$ of these special intervals.

(note that BITs assume one-indexing; it's just more elegant this way.)

In other words, we only require prefixes to be decomposable, not general ranges. The reason it's okay to do this is familiar (think prefix sums): every interval $[i, j]$ can be thought of as $[1, j]$ minus $[1, i-1]$. Therefore, computing the result for interval $[i, j]$ takes $\mathcal{O}(2 \lg n) = \mathcal{O}(\lg n)$ time.

When n is a power of two, you can see that the intervals are a subset of the intervals used by a segment tree. However, when n is not a power of two, the BIT first scales up n to the nearest power of two above n.

[24] Also called a Fenwick tree.

More formally, the special intervals used by BIT are as follows: $\{I_1, I_2, \ldots, I_n\}$, where $I_j = (j - \text{lowestBit}(j), j]$, and $\text{lowestBit}(j)$ is the lowest power of two 2^k contained in the binary representation of j.

It is instructive to check that the second property holds in this set of special intervals. Using this representation, we can now do point updates and range queries efficiently; the properties above guarantee efficiency!

However, unlike segment trees, BITs require our "addition" operation to have an inverse "subtraction", so that it can compute a general interval $[i, j]$ as the "difference" between $[1, j]$ and $[1, i-1]$. This is why segment trees are thought of as more flexible, since it doesn't have this requirement, and it can use any operation at all (as long as it is associative).

But BIT is still an okay alternative because it is very easy to iterate through the required special intervals! To illustrate, here is how you would decompose the prefix $[1, j]$ into a sum of special intervals (assuming 1-indexing):

```
void decompose_prefix(int j) {
    cout << "[1, " << j << "] can be decomposed into:";
    while (j > 0) {
        cout << " I_" << j;
        j -= lowestBit(j);
    }
    cout << endl;
}
```

And here is how you would iterate through all special intervals containing position j (assuming 1-indexing):

```
void intervals_containing(int j) {
    cout << "Position " << j << " is contained in the intervals:";
    while (j <= n) {
        cout << " I_" << j;
        j += lowestBit(j);
    }
    cout << endl;
}
```

And `lowestBit(j)` itself can be computed quickly with the following cute snippet: `j & -j` (where `&` is the bitwise AND operator).

Exercise 4.8. Explain why `j & -j` is the lowest power of two 2^k contained in j.

Hint: For this, you will need to understand two's complement on negative integers.

More details can be found in the TopCoder article on the same topic.

4.4.3 Implementation

The following is an implementation of the **inc** and **sum** operations. Note that **sum** has two flavors – if we provide one parameter p, it will get the sum of all elements indexed 0 to p, and if we provide two parameters a and b, we get the sum from index a to index b.

Also, note that the operations are 0-indexed from the perspective of the user, but internally, they are converted to 1-indexing (with `p+1`).

```cpp
class BIT {
private:
    vector<int> tree;
    int n;

public:
    BIT(int n): n(n), tree(n+1) {}  // initialize the tree with n+1 elements.

    void inc(int p, int v) {
        for(int i = p+1; i <= n; i += i & -i)
            tree[i] += v;
    }

    int sum(int p) {
        int v = 0;
        for(int i = p+1; i > 0; i -= i & -i)
            v += tree[i];
        return v;
    }

    int sum(int a, int b) {
        return sum(b+1) - sum(a);
    }
};
```

This implementation assumes that the array is initially empty. If it is not, then one can initialize the BIT by simply performing **inc** n times, one for each initial value. This runs in $\mathcal{O}(n \lg n)$ time.

> **Exercise 4.9.** Explain how to implement **create** in $\mathcal{O}(n)$ time instead of $\mathcal{O}(n \lg n)$ time, without using up additional memory, even temporarily.
>
> *Note:* Although it is possible, it is usually not done anyway because it is much more complicated than the naive method.

4.4.4 Multidimensional Binary-Indexed Tree

If our range is multidimensional, we can easily modify our implementation to support it by just adding more loops. For example, if we have a 2D range, we modify our functions as so.

```cpp
class BIT {
private:
    vector<vector<int>> tree;
    int n, m;

public:
    BIT(int n, int m): n(n), m(m), tree(n+1, vector<int>(m+1)) {}  // initialize the 2D tree
    ↪ with (n+1)*(m+1) elements.

    void inc(int x, int y, int v) {
        for(int i = x+1; i <= n; i += i&-i)
            for(int j = y+1; j <= m; j += j&-j)
                tree[i][j] += v;
```

```
    }

    int sum(int x, int y) {
        int v = 0;
        for(int i = x+1; i > 0; i -= i&-i)
            for(int j = y+1; j > 0; j -= j&-j)
                v += tree[i][j];
        return v;
    }

    int sum(int x0, int y0, int x1, int y1) {
        return sum(x1+1,y1+1) - sum(x1+1,y0) - sum(x0,y1+1) + sum(x0,y0);
    }
};
```

If the range has dimensions $n \times m$, then **inc** and **sum** have complexity $\mathcal{O}(\lg n \lg m)$.

Exercise 4.10. What does `tree[i][j]` represent in the 2D BIT?

4.4.5 Range Update, Point Query

Another variation of the use of Binary-Indexed Trees is one that handles *range updates* and *point queries*. This is in contrast to earlier implementations where we update single elements (point update) and query ranges.

If we add a constant element v to a single index i and keep track of the cumulative sum, then we effectively increase this cumulative sum by v from index i onwards. If we couple this by subtracting the same constant element v from a single index $j > i$, then we effectively only increase the elements from i to $j - 1$ by v. This is the principle behind the range updates.

The following is an implementation of the above algorithm. It implements two functions: `inc` which adds a constant v to a range $[a, b]$, and `get` which returns the value at index p.

```
class BIT {
private:
    vector<int> tree;
    int n;

    void inc(int p, int x) {
        for(int i = p+1; i <= n; i += i&-i)
            tree[i] += x;
    }

public:
    BIT(int n): n(n), tree(n+1) {}

    void inc(int a, int b, int v) {
        inc(a, v);
        inc(b+1, -v);
    }

    int get(int p) {
```

```
        int v = 0;
        for(int i = p+1; i > 0; i -= i&-i)
            v += tree[i];
        return v;
    }
};
```

A more advanced variation allows you to perform *range updates* and *range queries* too, by using *line functions* (see problem **C6**). But at this point, the implementation gets too complicated already, so the author recommends just using segment trees instead, which will be able to handle them easily in an upcoming module.

4.5 Sparse Tables

4.5.1 What does it do?

A **sparse table** is a data structure that preprocesses *static* elements of a range in $\mathcal{O}(n \lg n)$, and provides answers to some types of range queries in $\mathcal{O}(1)$. The most common queries are range minimum/maximum queries. Sparse tables are *static*, that is, they don't provide update functionality. However, unlike prefix sums, they can support operations like **min** which doesn't have an inverse operation.

4.5.2 Theory

Sparse Tables make use of Dynamic Programming to make its queries efficient. For the following example, let's say we're trying to get a range maximum query implementation. For this, we are given an array A of size n with initial values. Let's define a function $f(i, k)$ to be the minimum of the elements i to $i + 2^k - 1$, or more compactly $f(i, k) = \min_{i \le j < i+2^k} A_j$. Another way to look at it is that f operates on all ranges with lengths that are powers of two. We can express the function with the following recurrence.

$$f(i, 0) = \begin{cases} A_i & \text{if } 0 \le i < n \\ \infty & \text{otherwise} \end{cases}$$
$$f(i, k) = \min\Big(f(i, k-1),\ f(i + 2^{k-1}, k-1)\Big)$$

After preprocessing, we can answer range minimum queries in $\mathcal{O}(1)$. Let (a, b) be the query range we need to solve. Let k be the largest integer such that $2^k \le (b - a + 1)$, or in other words 2^k is the largest power of 2 that can "fit" inside the range of the query. Then we can answer the query by simply computing $\min\Big(f(a, k),\ f(b - 2^k + 1)\Big)$. This works because the two preprocessed range minimum values we're comparing covers the elements in the query range.

Note that we only need i up until $n - 1$ and k until $\lfloor \lg n \rfloor + 1$, giving us a worst-case time complexity of $\mathcal{O}(n \lg n)$.

Another way of looking at sparse tables is that it tries to find a set of special intervals such that every interval $[i, j]$ can be *covered* with at most two special intervals. These two special intervals may overlap, but this is fine for operations like **min** and **max** where operating on an element twice doesn't affect the result.[25] However, it is not suitable for operations like **sum**.

[25] also called an **idempotent** function.

Obviously, one can just declare *all* intervals as special intervals, but that's not usually good because there are *too many* intervals, specifically, $\Omega(n^2)$ of them. The beauty of sparse tables is that the set of special intervals is *sparse*, only $\mathcal{O}(n \lg n)$ special intervals, while still allowing $\mathcal{O}(1)$ answering of queries.

4.5.3 Implementation

An implementation of the above data structure is shown below.[26] Note that there is no update function.

```
#define INF 1000000001
class SparseTable {
public:
    vector<vector<int>> f;
    vector<int> lg;

    SparseTable() {}
    SparseTable(vector<int> A) {
        int n = A.size();

        // preprocess largest k such that 2^k <= i
        for(int i = 2; i <= n; i++) lg[i] = lg[i>>1] + 1;

        // computes f(k,i). note that k and i are reversed
        int m = lg[n];  // largest possible k
        f = vector<vector<int>>(m+1, vector<int>(n));
        for(int i = 0; i < n; i++)
            f[0][i] = A[i];
        for (int k = 1; k <= m; k++) {
            f[k] = f[k-1];  // initialize with the previous row
            for (int i = 0, j = 1<<k-1; j < n; i++, j++) {
                f[k][i] = min(f[k-1][i], f[k-1][j]);
            }
        }
    }

    int min(int a, int b) {
        int k = lg[b-a+1];  // largest 2^k contained in [a, b]
        return min(f[k][a], f[k][b-(1<<k)+1]);
    }
};
```

[26] Note that in the implementation, we preprocessed the proper k for each query length. The other option is to call the `log2` function, which can save us memory at the cost of higher overhead for queries.

5 Problems

Solve as many as you can! Ask me if anything is unclear.[27] In general, the harder problems will be worth more points, but this is only an approximation.

Problems with red points are considered more important.

5.1 Non-coding problems

No need to be overly formal in your answers; as long as you're able to convince me, it's fine!

Every exercise mentioned in the main text is worth [100★].

First timers, solve at least [1200★]. Veterans, solve at least [2000★].

N1 Apart from time complexity, space/memory complexity is also important – we can be as efficient as we can in terms of time, but we might exceed the memory limit if we're not careful. For the following data structures, what are the space complexities? Answer in Big $\mathcal{O}$ notation, e.g. $\mathcal{O}(n^6.9)$. Explain your answer.

(a) [80★] Linked Lists

(b) [80★] Binary Search Trees

(c) [80★] Binary Heaps

(d) [80★] Binary-Indexed Tree

(e) [80★] Segment Tree

(f) [80★] Sparse Table

N2 In the "doubling" scheme for vectors, if we try to insert an element when the vector has full capacity, it will reallocate to double the capacity. Since it will only reallocate a few times, we can say that the `push_back` function has a complexity of $\mathcal{O}(1)$ amortized.

(a) [150★] Consider the scheme where instead of doubling the capacity when full, we increase it by a fixed constant factor $c > 1$ instead (in other words, the new capacity is c times the previous). Will it still be $\mathcal{O}(1)$ amortized? If not, what is the amortized (average) complexity (perhaps in terms of c)? Explain your answer, and include your assumptions if any.

(b) [150★] Considering the vector resizing scheme where instead of doubling the capacity when full, we increase the size by a fixed value k. Will it still be $\mathcal{O}(1)$ amortized? If not, what is the amortized (average) complexity (perhaps in terms of k)? Explain your answer, and include your assumptions if any.

N3 [100★] Describe a data structure that implements the following ADT, known as the **disjoint sets** data type. A disjoint-sets data structure represents a set of disjoint sets, with the following operations:

- **disjointSets.create()**. Create a new set of disjoint sets s, initially empty.
- s**.makeSet(**v**)**. Create a new set containing only v. (assume that v doesn't exist yet in any set.)

[27] Especially for ambiguities! Otherwise, you might risk getting fewer points even if you *technically* answered the question correctly.

- s**.find(**v**)**. Return a "representative" element from the set containing v. The representative element must be the same for every set, regardless of the v used to find it. Thus, we can use this operation to check if two elements v and w are in the same set: simply compare s**.find(**v**)** and s**.find(**w**)**!
- s**.union(**v**,** w**)**. Merge/Unite the sets containing v and w (if they aren't already in the same set).

Note: This is a pretty simple exercise assuming you've read the Graphs 1 module. It's just restated in terms of abstract data types.

5.2 Coding problems

Class-based implementations are strongly recommended; the idea is to make the implementation easily reusable.[28] Making the implementation self-contained in a class makes it very easy for you to reuse, which is handy for your future contests!

First timers, solve at least [1500★]. Veterans, solve at least [3000★].

5.2.1 Implementation Problems

C1 [100★] Implement a BIT-like data structure with the following functions: **update**(p,k) modifies the p'th element (0-indexing) by changing the value to k, and **query**(i,j) returns the XOR of all elements from index i to j. It must do both in $\mathcal{O}(\lg n)$ time.

C2 [200★] Implement a deque using the "doubling" capacity technique shown in the vector implementation.[29] Your data structure must support the following functions: `push_back`, `pop_back`, `push_front`, `pop_front`, `get_element`[30], `front`, and `back` (the last two functions return the front and back elements, respectively). You're not allowed to use STL data structures. Make sure that all functions have a complexity of $\mathcal{O}(1)$ amortized.

C3 [150★] Implement a queue using two stacks. It must have the following functions: `push`, `pop`, and `front`. You may use STL stacks. All functions should have a complexity of $\mathcal{O}(1)$ amortized time.

C4 [150★] A *median queue* is a abstract data type that supports the following operations:

- **medianQueue.create()**. Returns a new, empty median queue m.
- m**.add**(v). Add a new element v to m.
- m**.median()**. Returns the median of m, or *both* median elemeents if the size of m is even.[31]

Implement a *median queue* using two priority queues. Each operation must take $\mathcal{O}(\lg n)$ time.

C5 [250★] Implement a sparse table for range sum queries. It should only have a single function – **.sum**(i,j) which returns the sum of all elements from index i to j. Initialization should take at most $\mathcal{O}(n \lg n)$ time, and each query should take $\mathcal{O}(1)$ time.

In addition, no subtraction of values is allowed.[32]

Note: Range sum queries can be answered with $\mathcal{O}(n)$ preprocessing and $\mathcal{O}(1)$ query using prefix sums, but that requires subtraction (inverse of addition). With this, we can't apply it to special operations that do not have an inverse. In fact, for this problem, the only assumption you need from the "addition" operation is that it is associative and has an identity.[33] You don't even need commutativity![34]

[28] See also: https://en.wikipedia.org/wiki/Modular_programming.

[29] Internally, STL Deques are a lot more complicated than this.

[30] Random access, similar to the [] operator we use to access indices; takes in one parameter which is the index. For those who are familiar to operator overloading, you may opt to overload the `[]` operator instead of the `get_element` function.

[31] The **median** is the middle element when it is sorted.

[32] This applies to preprocessed values of your sparse table. You are allowed to do other types of subtraction, such as subtraction of indices, etc.

[33] In *group theory* terms, the operation forms a **monoid** over the set of values.

[34] Such operations exist. An example is multiplication of 2×2 matrices, where the matrices are not necessarily invertible.

Hint: Remember that a sparse table is basically a set of $\mathcal{O}(n \lg n)$ *special intervals* such that every interval $[i, j]$ can be *covered* with at most two special intervals. Can you find something stronger, that is, a set of $\mathcal{O}(n \lg n)$ special intervals such that every interval $[i, j]$ can be decomposed into most two **disjoint** intervals?

C6 [300★] The goal here is to use a BIT to implement range **sum**s and range **inc**s.

A **line function** is a function of the form $x \to ax + b$ for some constants a and b.

One way of looking at a range **inc** is as follows. Let $s_1, s_2, \ldots, s_n$ be the list of prefix sums. Then the range **inc** from index i to index j with value v is equivalent to adding v to s_i, $2v$ to s_{i+1}, $3v$ to s_{i+2}, etc. In other words, it adds $(k-i+1) \cdot v$ to s_k for all $k \in [i, j]$. In other words, we're adding the line function $x \to vx + v(-i+1)$ in a range.

By building a BIT on top of a list of line functions, show how to handle range **sum**s and range **inc**s in $\mathcal{O}(\lg n)$.

Hint: You actually need two BITs, one for the as, and one for the bs.

Hint 2: You need point **sum**s and range **inc**s on the list of line functions.

5.2.2 Online Judge Problems

You can solve as many problems as you want. Each category will give you a certain number of points per problem solved. Submit directly to the specified online judges.

For those that were partially solved (such as Hacker Rank subtasks), you can opt for partial points by submitting your code in the submission bin at the end of the week. Note that the partial points may not be up to scale with the points of the subtask. If you have a partially solved problem that doesn't have subtasks, but you think you were getting near the solution, you can opt to submit it as well for consideration. For all cases, include the details as to how you tried to approach the problem, and any insights you've garnered (you can just add it as a comment in your code).

[**Easy**] These problems are simple applications of data structures.

S1.1 [50★] **Balanced Brackets:** https://www.hackerrank.com/contests/noi-ph-2019-preselection/challenges/balanced-brackets

S1.2 [50★] **Strange Race:** https://www.codechef.com/problems/GOC205

S1.3 [50★] **Kth Max Subarray:** https://www.codechef.com/problems/KTHMAX

S1.4 [50★] **Finding the Running Median:** https://www.hackerrank.com/contests/noi-ph-2019-preselection/challenges/find-the-running-median

---

[**Medium**] Some of these problems require a hint of creativity.

S2.1 [200★] **DNA Evolution:** https://codeforces.com/problemset/problem/827/C

S2.2 [200★] **Cyclic Shifts:** https://www.codechef.com/problems/MINSHIFT

S2.3 [200★] **Range Minimum Queries:** https://www.codechef.com/problems/ANDMIN

S2.4 [200★] **Nice SubSegments:** https://www.codechef.com/problems/SUBSGM

S2.5 [200★] **Perfect Subarrays:** https://www.codechef.com/problems/SUBARR

S2.6 [200★] **Devu and Manhattan Distance:** https://www.codechef.com/problems/MDIST

S2.7 [200★] **Another Query on Array Problem:** https://open.kattis.com/problems/queryonarray

S2.8 [200★] **Karen and Coffee:** https://codeforces.com/problemset/problem/816/B

S2.9 [200★] **Sagheer and Nubian Market:** https://codeforces.com/problemset/problem/812/C

S2.10 [200★] **Honey Heist:** https://open.kattis.com/problems/honeyheist

S2.11 [200★] **Triangle Triples:** https://projecteuler.net/problem=378

S2.12 [200★] **Factorial Array:** https://www.hackerrank.com/contests/noi-ph-2019-preselection/challenges/factorial-array

---

[**Hard**] Solving these problems require creative thinking and extra effort.

S3.1 [500★] **Ordered radicals:** https://www.hackerrank.com/contests/projecteuler/challenges/euler124

S3.2 [500★] **New Year and Old Subsequence:** https://codeforces.com/problemset/problem/750/E

S3.3 [500★] **Möbius function and intervals:** https://projecteuler.net/problem=464

S3.4 [500★] **Confidential Message:** https://www.hackerrank.com/contests/noi-ph-2019-preselection/challenges/confidentialegassem-8 (sqrt decomposition required)

S3.5 [500★] **Chef has a New Kitchen:** https://www.codechef.com/problems/CHEFNKCH

S3.6 [500★] **Segment Tree:** https://www.codechef.com/problems/SEGTREE2

S3.7 [500★] **Small Cubes:** https://www.hackerrank.com/contests/noi-ph-2019-preselection/challenges/small-cubes

S3.8 [500★] **Instachallenge:** https://www.hackerrank.com/contests/noi-ph-2019-preselection/challenges/instachallenge

Acknowledgment

Most of the material is based on the Data Structures materials written by Josh Quinto.

A Stack vs Heap

A running C++ program splits the available memory into two parts, the **stack** memory and the **heap** memory.[35] We will go over them in more detail in a later module, but for now, I'll give a quick overview, since this will be slightly relevant when we implement self-referential/recursive data structures.

A real-life *stack* is just a bunch of objects sitting atop of each other. One property of a stack is that you can only ever interact with its *top*. You can take the topmost object, or put something at the top, but you can't mess around with the middle or bottom ones. The **stack memory** is kinda like that. Whenever the C++ program needs some memory from the stack memory, it only allocates the first available portion. Also, when it wants to free up memory, it can only free up the most-recently allocated ones.

When you declare a variable such as `CustomClass instance;`, the program allocates it onto the stack memory and then runs the constructor. Thereafter, it stays in stack memory until `instance` is no longer needed; for example, if you've returned from the function call containing it, or if it's in an inner scope (such as an `if` block) and you've just exited that block. When that happens, the program automatically calls the destructor. The program automatically does all this for you, so you don't have to manage the stack and call the destructor yourself.

As you can imagine, the stack memory is what allows the program to easily keep track of local variables, including local variables in function calls—every time you call a function, the program can simply use more memory from the stack. Indeed, you can think of calling a function as pushing onto the stack, and returning from a function as popping the top of the stack. This is why each function call to `fibonacci(n)` gets its own copy of `n` at every level of the recursive call!

Unfortunately, you can't use the stack for all purposes. For example, you can't initialize *dynamically-sized* structures this way. You can only allocate fixed-sized things. If you need dynamic allocation, then you need to use the **heap memory**. For example, if you need a dynamically-resizing list or tree structure, then you'll have to use the heap.

To allocate an instance of `CustomClass` on the heap, you have to declare it as `CustomClass*` (with an asterisk) and then use the `new` keyword. This calls the constructor as well. To free up something you allocated (and then call the destructor), use the `delete` keyword. Also, to access the attributes, you use the syntax '`->`' instead of '`.`' (you don't need to know why for now).

Here's an example:

```
int main() {
    CustomClass* instance1 = new CustomClass;
    CustomClass* instance2 = new CustomClass(1, "Hello World", -3.54);
    instance2->foo();
    cout << instance2->bar() << '\n';
    delete instance2;  // deallocates from the heap
}
```

Note that you need to use `new` even if you're using the empty constructor. Also, note that you can only `delete` something you allocated with `new`.

Unlike with the stack memory, the stuff allocated on the heap stay even after you return from a function call.[36]

Now you might be thinking, "but I've been using C++'s `vector` class all this time like a

[35] Technically, there are other parts, but we won't need them for now.

[36] This is why the program doesn't call the destructor automatically; you do that by doing `delete` yourself.

normal variable. It's dynamically sized, yet I don't have to do this `new` and * thing!" Good question! The reason is that the programmers who wrote the `vector` class were sneaky and clever: the `vector` class does the heap allocation and deallocation for you behind the scenes! When you declare a `vector` normally (on the stack), it does all this heap allocation in the constructor, and heap deallocation in the destructor, all without you knowing. So for all practical purposes, you can pretend that it's on the stack!

Another thing you might want to know is that `CustomClass*` is considered a different type than `CustomClass`. Whereas a `CustomClass` variable contains an instance of the `CustomClass` class, a `CustomClass*` variable does not; it's only a reminder that there's a `CustomClass` somewhere in the heap, and the `CustomClass*` variable is basically your "access" to it. That's why you use `->something` instead of `.something`—you're not directly getting the "`something`" from your variable, but you're getting the "`something`" of some object somewhere in the heap (denoted by your variable).

In other words, a variable of type `CustomClass*` merely *points* to something of type `CustomClass`. For this reason, `CustomClass*` is called a *pointer* type.

By the way, if you want to allocate or deallocate arrays on the heap, you use `new[]` and `delete[]` instead. For example:

```
int *A = new int[100]; // declare array of 100 integers
delete[] A; // delete array
```

So when do you use the heap memory at all instead of the stack memory? While stack allocation is pretty fast, there are a few issues:

1. When your class is large, passing instances of it around to functions is expensive, since the program has to copy it onto the stack every time you do so. For example, if you have something like

```
class LargeClass {
public:
    int array[1000];
    char str[100];
    LargeClass() {}
};

void foo(LargeClass x) {
    ...
}
```

   then every time you call `foo`, the whole class has to be copied over.

   In contrast, a `LargeClass*` variable is just a pointer, and it's pretty lightweight: passing it around is basically as fast as passing a `long long` around! This is because the `LargeClass` is in the heap, not in the stack.

   Note: Passing a regular `vector` is expensive, even though I said above that much of the internal contents are on the heap! The reason is that the implementors of `vector` really tried their hard to mimic the behavior of regular variables on the stack, so they coded it in such a way that when a `vector` variable is copied to another `vector` variable, the stuff on the heap is also automatically copied! They implement it this way so that you can really, truly pretend that a `vector` as being fully in the stack for all practical

purposes.

2. Suppose you want to create a class that has an attribute that you only want to be there *optionally*, i.e., sometimes it's there, sometimes it's not. For example, say you want a `Person` class and you want an optional `pet` attribute:

```
class Dog {
public:
    string name;
};
class Person {
public:
    string name;
    Dog pet;  // I want this to be optional
};
```

Here, `pet` isn't optional yet. However, if we want it to be optional, then we can't seem to do it!

One thing we can do is just *pretend* that the `pet` is not there when we don't need it. The issue with this is that the program still allocates space for a `Dog` even if you aren't gonna use it, which is fine in some cases, but when `Dog` is large, then this is an issue since we're wasting resources.

The solution is to use the heap, and pointers:

```
class Person {
public:
    string name;
    Dog* pet;
};
```

Then if we want to initialize a pet `Dog` for a `Person`, we just allocate in the heap via `new`. And if we don't want them to have a pet, then we can simply initialize it with the *null pointer* (a pointer that points to *nothing*), written in C++ as `nullptr`.

```
class Person {
public:
    string name;
    Dog* pet;
    Person(string n) {
        name = n;
        pet = nullptr;
    }
    Person(string n, string petn) {
        name = n;
        pet = new Dog(petn);
    }
};
```

This is all the more important if your optional attribute has a *recursive* type, such as in the following example:

```cpp
class Person {
public:
    string name;
    Person* spouse;
};
```

In this case, you *have* to use pointers, otherwise the compiler will complain!

B Generic data types in C++

When implementing a *container-type* structure, you usually have to decide early on what type of elements it will contain. However, sometimes, you want your container to be generic enough, that it can contain *any type* you want. To implement such containers as C++ classes, we can use *templates*.[37]

Let's look at a normal implementation of a doubly linked list:

```
class LLNode {
public:
    char v;
    LLNode *prev, *next;

    LLNode() {}
    LLNode(char v, LLNode *prev, LLNode *next): v(v), prev(prev), next(next) {}
};

class LinkedList {
public:
    LLNode *head = nullptr, *tail = nullptr;
    int length = 0;

    LinkedList() {}
    LinkedList(const vector<char>& init) {
        for (char v : init) push_back(v);
    }

    void insert_after(LLNode *node, char v) {
        length++;

        LLNode *bago;
        if (node == nullptr) {  // insert to front
            bago = head = new LLNode(v, node, head);
        } else {
            bago = node->next = new LLNode(v, node, node->next);
        }
        if (bago->next != nullptr) bago->next->prev = bago;
        if (tail == node) tail = bago;
    }

    void remove(LLNode *node) {  // removes the given node
        length--;

        if (node->next) node->next->prev = node->prev;
        if (node->prev) node->prev->next = node->next;
        if (head == node) head = node->next;
        if (tail == node) tail = tail->prev;

        delete node;
    }

    void push_back(char v) { insert_after(tail, v); }
    void push_front(char v) { insert_after(nullptr, v); }
    void pop_back() { remove(tail); }
    void pop_front() { remove(head); }
    char back() { return tail->v; }
```

[37] Java programmers have *generics* that does a similar role. Python programmers don't encounter this issue because types don't have to be declared.

```
    char front() { return head->v; }
    int size() { return length; }
};
```

This contains **char**s as values. However, the linked list doesn't really have anything to do with **char**s; it's just what we arbitrarily chose to be what the linked list will hold. So if we wanted it to contain **int**s, we just replace **char** with **int**s.

To convert this into generic style, we simply use a *type parameter* `T` using the following syntax:

```
template<class T>
class LLNode {
public:
    T v;
    LLNode<T> *prev, *next;

    LLNode() {}
    LLNode(T v, LLNode<T> *prev, LLNode<T> *next): v(v), prev(prev), next(next) {}
};

template<class T>
class LinkedList {
public:
    LLNode<T> *head = nullptr, *tail = nullptr;
    int length = 0;

    LinkedList() {}
    LinkedList(const vector<T>& init) {
        for (T v : init) push_back(v);
    }

    void insert_after(LLNode<T> *node, T v) {
        length++;

        LLNode<T> *bago;
        if (node == nullptr) {  // insert to front
            bago = head = new LLNode<T>(v, node, head);
        } else {
            bago = node->next = new LLNode<T>(v, node, node->next);
        }
        if (bago->next != nullptr) bago->next->prev = bago;
        if (tail == node) tail = bago;
    }

    void remove(LLNode<T> *node) {  // removes the given node
        length--;

        if (node->next) node->next->prev = node->prev;
        if (node->prev) node->prev->next = node->next;
        if (head == node) head = node->next;
        if (tail == node) tail = tail->prev;

        delete node;
    }

    void push_back(T v) { insert_after(tail, v); }
    void push_front(T v) { insert_after(nullptr, v); }
    void pop_back() { remove(tail); }
    void pop_front() { remove(head); }
    T back() { return tail->v; }
```

```
    T front() { return head->v; }
    int size() { return length; }
};
```

Notice that we really mostly just replaced all `char` with `T` (and updated the declarations of some variables).

To create a linked list of `char`s, we simply declare our variable as `LinkedList<char> x;`. But we can also use a list of `int`s as `LinkedList<int> y;`. We can even create a linked list of linked lists, something like `LinkedList<LinkedList<int>> z;`!

In fact, you're probably familiar with the syntax for using it, since this is how the C++ library structures are implemented!

We can also implement a generic segment tree containing *anything* that supports addition, as follows:

```
template<class T>
class SegmentTree {
private:
    int i, j; // range represented by this node
    T v; // total at this range
    SegmentTree<T> *left, *right;

    SegmentTree(const vector<T>& init): SegmentTree(init, 0, init.size() - 1) {} // call the
    ↪  other constructor

    SegmentTree(const vector<T>& init, int i, int j): i(i), j(j) {  // initialize i and j
        if (i == j) {
            m = init[i];
            left = right = nullptr;
        } else {
            int k = (i + j) / 2;
            left = new SegmentTree(init, i, k);
            right = new SegmentTree(init, k + 1, j);
            combine();
        }
    }

    void combine() {
        v = left->v + right->v;
    }

    void set(int I, T V) {
        if (i <= I && I <= j) {
            if (left) {
                left->set(I, V);
                right->set(I, V);
                combine();
            } else {  // leaf node
                v = V;
            }
        }
    }

    T sum(int I, int J) {
        if (I <= i && j <= J) {  // completely contained
            return m;
        } else if (J < i || j < I) {  // completely disjoint
            return T();  // "identity"
        } else {  // partially overlaps
```

```
            return left->sum(I, J) + right->sum(I, J);
        }
    }
};
```

This assumes that `T()` returns the identity element. Other than that, the implementation only assumes that `+` is associative. It doesn't require commutativity.

We can now use it as `SegmentTree<int>`, or `SegmentTree<double>`, or `SegmentTree<ll>`, etc.