
FREE eBook
LEARNING
data-structures
Free unaffiliated eBook created from
Stack Overflow contributors.
#data-
structures

Table of Contents

About

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: data-structures

It is an unofficial and free data-structures ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official data-structures.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapter 1: Getting started with data-structures

Remarks

This section provides an overview of what data-structures is, and why a developer might want to use it.

It should also mention any large subjects within data-structures, and link out to the related topics. Since the Documentation for data-structures is new, you may need to create initial versions of those related topics.

Examples

Intro to Data Structures

A data structure is a way of organizing and storing information.

Let a "Hello, World!" string be the information that we need to organize and store in byte-addressable memory.

Each ASCII character requires 7 bits of storage. Most systems reserve 8 bits (1 byte) for each character, so each character in "Hello, World!" is stored in an individual byte-sized unit of memory, one after another, consecutively.

We need a single reference to our string even though it spans multiple memory addresses, so we use the address of the first character in the string, 'H'. Every other character can be accessed at *the address of 'H' + the index of that character* using zero-indexed characters.

We want to print our string, "Hello, World!" We know its address in memory, which we supply to the print function, but how does the print function know to stop printing consecutive memory locations? One common approach is to append the null character, '\0', to the string. When the print function encounters the null character, it knows that it has reached the end of the string.

We have defined a way of organizing and storing our string, i.e. a data structure! This very simple data structure is a null-terminated character array, which is one way to organize and store a string.

Array : A Simple Data Structure

> An Array Data Structure is used to store similar objects (or data values) in a contiguous block of memory. Array Data Structure has fixed size, which determines the number of data values that can be stored in it.

Array : The C++ Way

In C++ Programming Language, we can declare a static array as follow

```
int arrayName[100];
```

Here we have declared an array named as "arrayName" which can store up to 100 values, all of which are of the same type, that is an integer.

Now, we will discuss some advantages and disadvantages of this Data Structure

1. We can access Data Values stored in Array in Constant Time, that is the time complexity is O(1). So if we want to access the data value stored at the i-th position, we need not start from starting position and move up to the i-th position, but we can directly jump to i-th position thus saving computing time.
2. Inserting an element in middle of an array is not an efficient task. Suppose we want to add a new element in the array at i-th position, then we need to first move all element at (i-th) and (i+1 th) position to create space for new element. Example : `1 4 2 0` is an array with 4 elements , now we want to insert 3 at 2nd position then we need to move 4,2 and 0 one position further to create space for 3.

```
1 3 4 2 0
```

3. Similar to insertion of the element, deletion of an element from an i-th position in an array is also not efficient since we need to move all elements ahead of the deleted element by 1 block in order to fill the vacant space created by the deleted element.

These are 3 simple characteristics of an array, Here you might believe that array is not an efficient Data Structure, but in practice, Advantage of an array can outweight its dis-advantages. This largely depend upon the kind of purpose you want to serve, it might be possible that you don't want to insert or delete element as often as you want to access them, in that case, an array is an absolutely perfect Data Structure.

The sole purpose of introducing this data structure is to make sure you simply don't choose Data Structure based on the number of advantages and disadvantage, but you should always try to analyse importance of Data Structure by considering your problem in mind, for example, if you will be spending a lot of time accessing data values as compared to inserting or deleting them, in that case, we need to give more weight to advantage over disadvantage.

Read Getting started with data-structures online: https://riptutorial.com/data-structures/topic/2378/getting-started-with-data-structures

Chapter 2: Binary Heap

Introduction

A binary heap is a complete binary tree which satisfies the heap ordering property. The ordering can be one of two types: the min-heap property: the value of each node is greater than or equal to the value of its parent, with the minimum-value element at the root.

Examples

Example

Min-Heap

```
        1
      /   \
     2     3
    / \   / \
   4   5 6   7
```

The above tree is a Min-Heap since the root is the minimum among all the nodes present in the tree.The same property is followed by all the nodes in the tree.

Max-Heap

```
        7
      /   \
     6     5
    / \   / \
   4   3 2   1
```

The above tree is a Max-Heap since the root is the maximum among all the nodes present in the tree.The same property is followed by all the nodes in the tree.

Operations Supported by Min-Heap

1. **getMin()** - It return the root element.Since it is the first element in an array we can retrieve the minimum element in `O(1)`
2. **extractMin()** - It removes the minimum element from the heap.Since after removing the element the tree must satisfy the Min-Heap property so a operation (**heapifying**) is performed to maintain the tree property.This takes `O(logn)`
3. **dereaseKey()** - It decreases the value of the key.Time complexity for this operation is `O(logn)`
4. **insert()** - The key is always inserted at the end of the tree.If the added key doesn't follow the heap property than we need to percolate up so that the tree satisfies the heap property.This

step takes takes `O(logn)` time.

5. **delete()** - This steps takes `O(logn)` time.For deleting a key we first need to decrease the key value to a minimum value and then extract this minimum value.

Application of Heap

1. Heap Sort
2. Priority Queue

Heap is used many of the graph algorithms like `Dijkstra's Shortest Path` and `Prim's Minimum Spanning Tree`.

Implementation in Java

```
public class MinHeap {

    int hArr[];
    int capacity;
    int heapSize;

    public MinHeap(int capacity){
        this.heapSize = 0;
        this.capacity = capacity;
        hArr = new int[capacity];
    }

    public int getparent(int i){
        return (i-1)/2;
    }

    public int getLeftChild(int i){
        return 2*i+1;
    }

    public int getRightChild(int i){
        return 2*i+2;
    }

    public void insertKey(int k){
        if(heapSize==capacity)
            return;

        heapSize++;
        int i = heapSize-1;
        hArr[i] = k;

        while(i!=0 && hArr[getparent(i)]>hArr[i]){
            swap(hArr[i],hArr[getparent(i)]);
            i = getparent(i);
        }
    }

    public int extractMin(){
        if(heapSize==0)
            return Integer.MAX_VALUE;

        if(heapSize==1){
```

```
            heapSize--;
            return hArr[0];
        }

        int root = hArr[0];
        hArr[0] = hArr[heapSize-1];
        heapSize--;
        MinHeapify(0);

        return root;
    }

    public void decreaseKey(int i , int newVal){
        hArr[i] = newVal;
        while(i!=0 && hArr[getparent(i)]>hArr[i]){
            swap(hArr[i],hArr[getparent(i)]);
            i = getparent(i);
        }
    }

    public void deleteKey(int i){
        decreaseKey(i, Integer.MIN_VALUE);
        extractMin();
    }

    public void MinHeapify(int i){
        int l = getLeftChild(i);
        int r = getRightChild(i);
        int smallest = i;
        if(l<heapSize && hArr[l] < hArr[i])
            smallest = l;
        if(l<heapSize && hArr[r] < hArr[smallest])
            smallest = r;

        if(smallest!=i){
            swap(hArr[i], hArr[smallest]);
            MinHeapify(smallest);
        }
    }

    public void swap(int x, int y){
        int temp = hArr[x];
        hArr[x] = hArr[y];
        hArr[y] = temp;
    }
}
```

Read Binary Heap online: https://riptutorial.com/data-structures/topic/10799/binary-heap

Chapter 3: Binary Search Tree

Examples

Creating a Node in BST

The Binary Search Tree (BST) is a hierarchical data structure with a single pointer to the root node.

The Node in the BST generally contains "items" (such as numbers or names) for fast look up. Each node has at-most two children (left and right). Every node is organized by some key data field. For every node in BST its key is greater than left child's key and less than right child's key

A typical structure of node (which stores an integer) would be

```
struct bst_node {
    int item;
    bst_node* left;
    bst_node* right;
};
```

There will be only one root node of BST. The root node can be created by

```
bst_node* root = NULL;
root = (bst_node*) malloc(sizeof(bst_node));
```

To set item key of root to 10.

```
root->item = 10;
```

inserting a node into binary search tree

```
struct tree{
    int a;
    tree* right;
    tree* left;
};
tree* root=NULL;
void insert(tree*& in, int b){
        if(in){
            if(in->a<b)
                    insert(in->right,b);
            else if(in->a>b)
                    insert(in->left,b);
            else
                cout<<"the value is already in the tree."<<endl;
        }else{
            tree* temp = new tree;
            temp->a=b;
            temp->right=NULL;
```

```
            temp->left=NULL;
            in=temp;
        }
}
```

Read Binary Search Tree online: https://riptutorial.com/data-structures/topic/6161/binary-search-tree

Chapter 4: Deque(Double ended queue)

Examples

Insertion and Deletion from both front and end of the Queue

Read Deque(Double ended queue) online: https://riptutorial.com/data-structures/topic/8849/deque-double-ended-queue-

Chapter 5: Graph traversals

Introduction

All the algorithms related to Graph traversals. Their complexities, both runtime and space

Examples

Depth First Search

Depth First Traversal (or Search) for a graph is similar to Depth First Traversal of a tree. The only catch here is, unlike trees, graphs may contain cycles, so we may come to the same node again. To avoid processing a node more than once, we use a boolean visited array.

Below algorithm presents the steps for graph traversal using DFS:

Algorithm DFS(v);

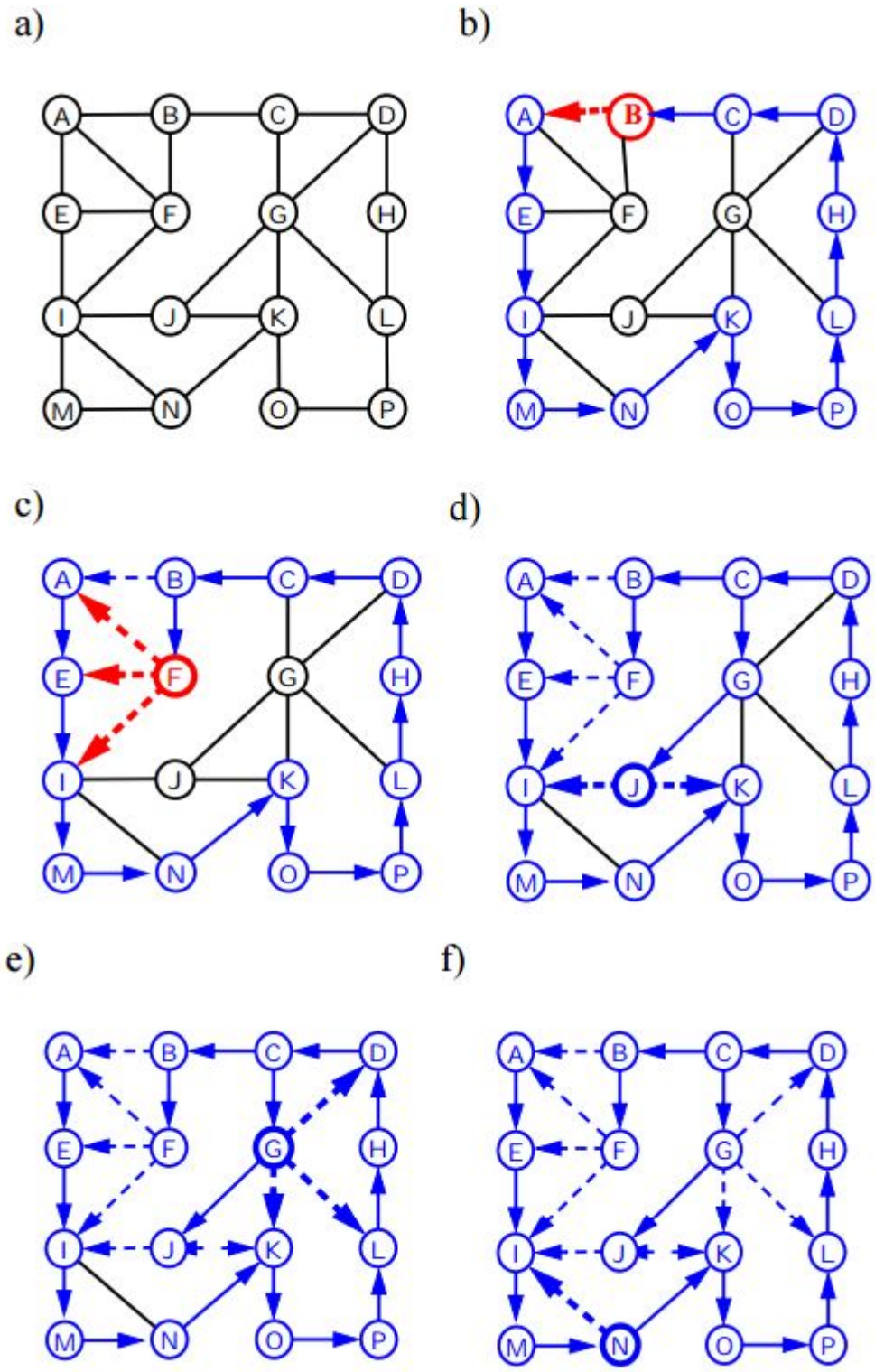
Input: A vertex v in a graph

Output: A labeling of the edges as “discovery” edges and “backedges”

```
for each edge e incident on v do

    if edge e is unexplored then

    let w be the other endpoint of e
    if vertex w is unexplored then
        label e as a discovery edge
        recursively call DFS(w)
    else
    label e as a backedge
```



Breadth First Search

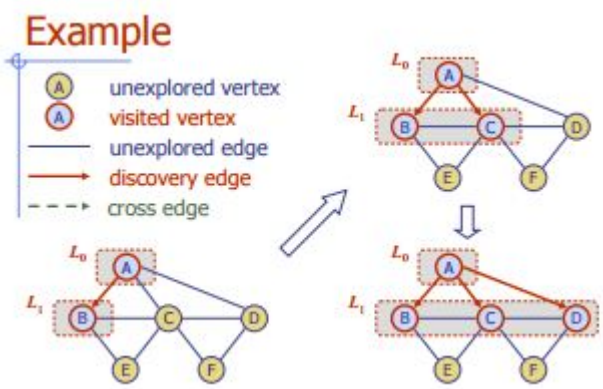
Algorithm BFS(G)

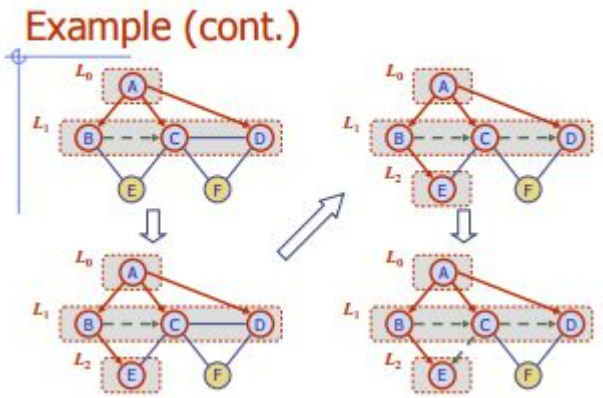
Input graph **G**

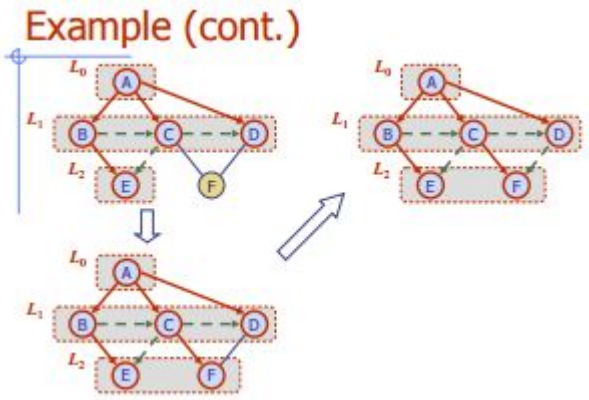
Output labeling of the edges and partition of the vertices of **G**

```
    for all u ∈ G.vertices()
        setLabel(u, UNEXPLORED)
```

```
    for all e ∈ G.edges()
    setLabel
    (e, UNEXPLORED)
    for all v ∈ G.vertices()
        if getLabel(v) = UNEXPLORED
            BFS(G, v)
```







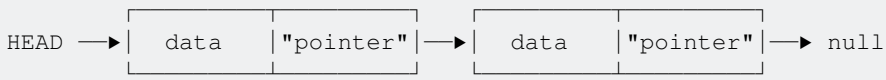
Read Graph traversals online: https://riptutorial.com/data-structures/topic/8688/graph-traversals

Chapter 6: Linked List

Examples

Introduction to Linked Lists

A linked list is a linear collection of data elements, called nodes, which are linked to other node(s) by means of a "pointer." Below is a singly linked list with a head reference.



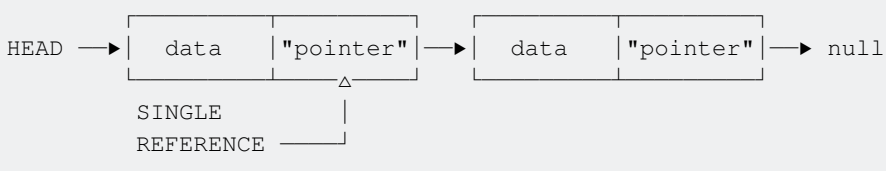
There are many types of linked lists, including singly and doubly linked lists and circular linked lists.

Advantages

- Linked lists are a dynamic data structure, which can grow and shrink, allocating and deallocating memory while the program is running.
- Node insertion and deletion operations are easily implemented in a linked list.
- Linear data structures such as stacks and queues are easily implemented with a linked list.
- Linked lists can reduce access time and may expand in real time without memory overhead.

Singly Linked List

Singly Linked Lists are a type of linked list. A singly linked list's nodes have only one "pointer" to another node, usually "next." It is called a singly linked list because each node only has a single "pointer" to another node. A singly linked list may have a head and/or tail reference. The advantage of having a tail reference are the `getFromBack`, `addToBack`, and `removeFromBack` cases, which become O(1).



Example Code in C

Example Code in Java, with unit tests - singly linked list with head reference

XOR Linked List

A **XOR Linked list** is also called a **Memory-Efficient Linked List**. It is another form of a doubly

linked list. This is highly dependent on the **XOR** logic gate and its properties.

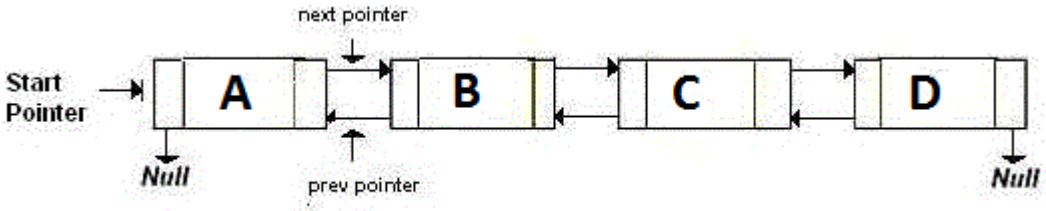
Why is this called the Memory-Efficient Linked List?

This is called so because this uses less memory than a traditional doubly linked list.

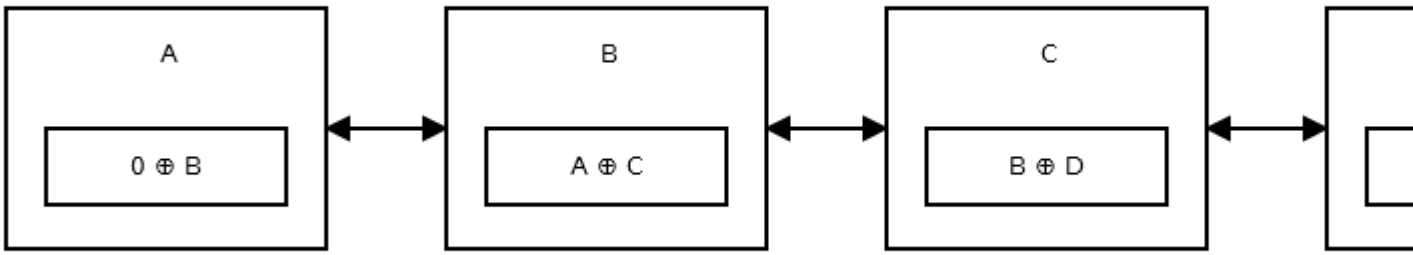
Is this different from a Doubly Linked List?

Yes, it is.

A **Doubly-Linked List** is storing two pointers, which point at the next and the previous node. Basically, if you want to go back, you go to the address pointed by the `back` pointer. If you want to go forward, you go to the address pointed by the `next` pointer. It's like:



A **Memory-Efficient Linked List**, or namely the **XOR Linked List** is having only one pointer instead of two. This stores the previous address (addr (prev)) **XOR** (^) the next address (addr (next)). When you want to move to the next node, you do certain calculations, and find the address of the next node. This is the same for going to the previous node. It is like:



How does it work?

To understand how the linked list works, you need to know the properties of XOR (^):

```
|-------------|------------|------------|
|    Name     |   Formula  |    Result  |
|-------------|------------|------------|
| Commutative |    A ^ B   |    B ^ A   |
|-------------|------------|------------|
| Associative | A ^ (B ^ C)| (A ^ B) ^ C|
|-------------|------------|------------|
| None (1)    |    A ^ 0   |     A      |
|-------------|------------|------------|
| None (2)    |    A ^ A   |     0      |
|-------------|------------|------------|
| None (3)    | (A ^ B) ^ A|     B      |
|-------------|------------|------------|
```

Now let's leave this aside, and see what each node stores.

The first node, or the **head**, stores `0 ^ addr (next)` as there is no previous node or address. It looks like this.

Then the second node stores `addr (prev) ^ addr (next)`. It looks like this.

The **tail** of the list, does not have any next node, so it stores `addr (prev) ^ 0`. It looks like this.

Moving from Head to the next node

Let's say you are now at the first node, or at node A. Now you want to move at node B. This is the formula for doing so:

```
Address of Next Node = Address of Previous Node ^ pointer in the current Node
```

So this would be:

```
addr (next) = addr (prev) ^ (0 ^ addr (next))
```

As this is the head, the previous address would simply be 0, so:

```
addr (next) = 0 ^ (0 ^ addr (next))
```

We can remove the parentheses:

```
addr (next) = 0 ^ 0 addr (next)
```

Using the `none (2)` property, we can say that `0 ^ 0` will always be 0:

```
addr (next) = 0 ^ addr (next)
```

Using the `none (1)` property, we can simplify it to:

```
addr (next) = addr (next)
```

You got the address of the next node!

Moving from a node to the next node

Now let's say we are in a middle node, which has a previous and next node.

Let's apply the formula:

```
Address of Next Node = Address of Previous Node ^ pointer in the current Node
```

Now substitute the values:

```
addr (next) = addr (prev) ^ (addr (prev) ^ addr (next))
```

Remove Parentheses:

```
addr (next) = addr (prev) ^ addr (prev) ^ addr (next)
```

Using the `none (2)` property, we can simplify:

```
addr (next) = 0 ^ addr (next)
```

Using the `none (1)` property, we can simplify:

```
addr (next) = addr (next)
```

And you get it!

Moving from a node to the node you were in earlier

If you aren't understanding the title, it basically means that if you were at node X, and have now moved to node Y, you want to go back to the node visited earlier, or basically node X.

This isn't a cumbersome task. Remember that I had mentioned above, that you store the address you were at in a temporary variable. So the address of the node you want to visit, is lying in a variable:

```
addr (prev) = temp_addr
```

Moving from a node to the previous node

This isn't the same as mentioned above. I mean to say that, you were at node Z, now you are at node Y, and want to go to node X.

This is nearly same as moving from a node to the next node. Just that this is it's vice-versa. When you write a program, you will use the same steps as I had mentioned in the moving from one node to the next node, just that you are finding the earlier element in the list than finding the next element.

Example Code in C

```
/* C/C++ Implementation of Memory efficient Doubly Linked List */
#include <stdio.h>
#include <stdlib.h>

// Node structure of a memory efficient doubly linked list
struct node
{
    int data;
    struct node* npx;  /* XOR of next and previous node */
};

/* returns XORed value of the node addresses */
struct node* XOR (struct node *a, struct node *b)
{
    return (struct node*) ((unsigned int) (a) ^ (unsigned int) (b));
}

/* Insert a node at the begining of the XORed linked list and makes the
   newly inserted node as head */
void insert(struct node **head_ref, int data)
{
    // Allocate memory for new node
    struct node *new_node  = (struct node *) malloc (sizeof (struct node) );
    new_node->data = data;

    /* Since new node is being inserted at the begining, npx of new node
       will always be XOR of current head and NULL */
    new_node->npx = XOR(*head_ref, NULL);

    /* If linked list is not empty, then npx of current head node will be XOR
       of new node and node next to current head */
    if (*head_ref != NULL)
    {
        // *(head_ref)->npx is XOR of NULL and next. So if we do XOR of
        // it with NULL, we get next
        struct node* next = XOR((*head_ref)->npx,  NULL);
        (*head_ref)->npx = XOR(new_node, next);
    }

    // Change head
    *head_ref = new_node;
}

// prints contents of doubly linked list in forward direction
void printList (struct node *head)
{
    struct node *curr = head;
    struct node *prev = NULL;
    struct node *next;

    printf ("Following are the nodes of Linked List: \n");

    while (curr != NULL)
    {
        // print current node
        printf ("%d ", curr->data);
```

```
        // get address of next node: curr->npx is next^prev, so curr->npx^prev
        // will be next^prev^prev which is next
        next = XOR (prev, curr->npx);

        // update prev and curr for next iteration
        prev = curr;
        curr = next;
    }
}

// Driver program to test above functions
int main ()
{
    /* Create following Doubly Linked List
       head-->40<-->30<-->20<-->10   */
    struct node *head = NULL;
    insert(&head, 10);
    insert(&head, 20);
    insert(&head, 30);
    insert(&head, 40);

    // print the created list
    printList (head);

    return (0);
}
```

The above code performs two basic functions: insertion and transversal.

- **Insertion:** This is performed by the function `insert`. This inserts a new node at the beginning. When this function is called, it puts the new node as the head, and the previous head node as the second node.
- **Traversal:** This is performed by the function `printList`. It simply goes through each node and prints out the value.

Note that XOR of pointers is not defined by C/C++ standard. So the above implementation may not work on all platforms.

References

- https://cybercruddotnet.wordpress.com/2012/07/04/complicating-things-with-xor-linked-lists/
- http://www.ritambhara.in/memory-efficient-doubly-linked-list/comment-page-1/
- http://www.geeksforgeeks.org/xor-linked-list-a-memory-efficient-doubly-linked-list-set-2/

Note that I have taken a lot of content from my own answer on main.

Skip List

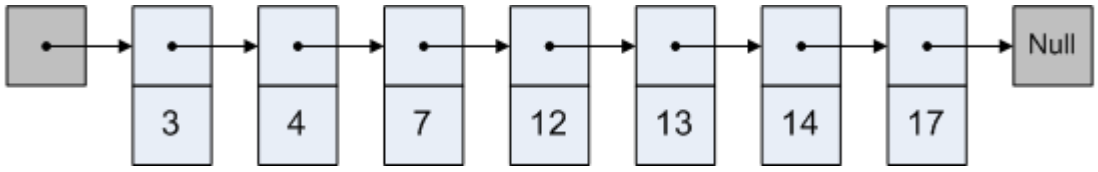
Skip lists are linked lists that allow you to skip to the correct node. This is a method which is way

more fast than a normal singly linked list. It is basically a singly linked list but the pointers not going from one node to the next node, but skipping few nodes. Thus the name "Skip List".

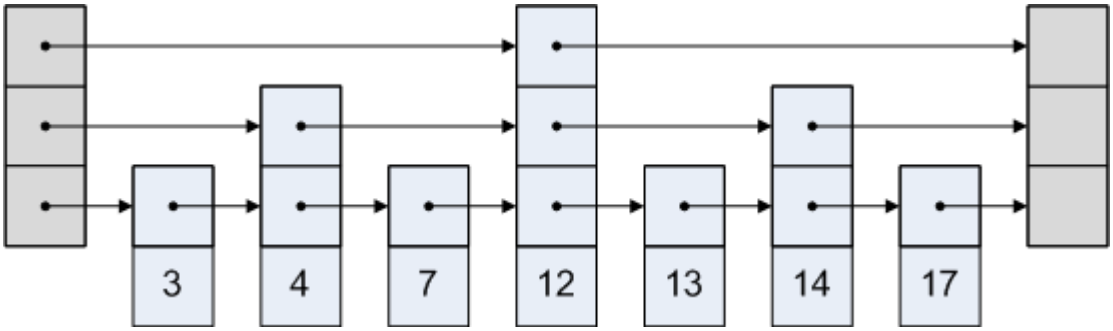
Is this different from a Singly Linked List?

Yes, it is.

A Singly Linked List is a list with each node pointing to the next node. A graphical representation of a singly linked list is like:



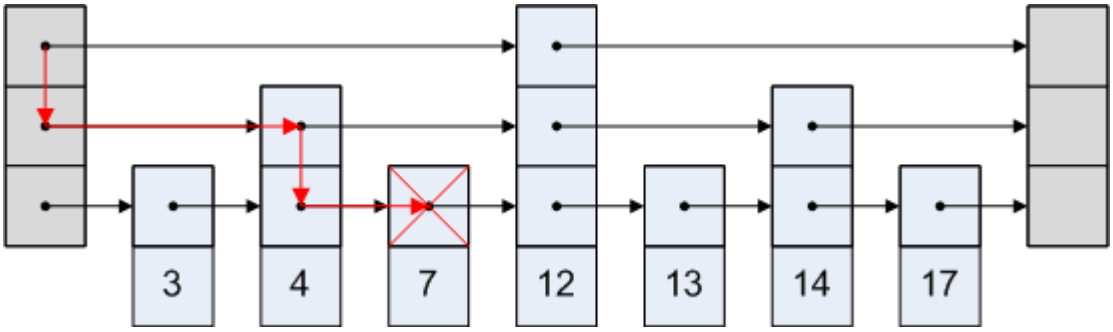
A Skip List is a list with each node pointing to a node which might or might not be after it. A graphical representation of a skip list is:



How does it work?

The Skip List is simple. Let's say we want to access node 3 in the above image. We can't take the route of going from the head to the fourth node, as that is after the third node. So we go from the head to the second node, and then to the third one.

A graphical representation is like:

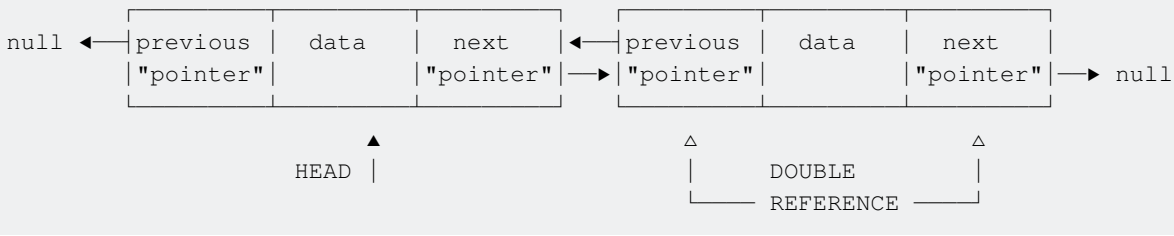


References

- http://igoro.com/archive/skip-lists-are-fascinating/

Doubly Linked List

Doubly Linked Lists are a type of linked list. A doubly linked list's nodes have two "pointers" to other nodes, "next" and "previous." It is called a double linked list because each node only has two "pointers" to other nodes. A doubly linked list may have a head and/or tail pointer.



Doubly linked lists are less space efficient than singly linked lists; however, for some operations, they offer significant improvements in time efficiency. A simple example is the `get` function, which for a doubly linked list with a head and tail reference will start from head or tail depending on the index of the element to get. For an `n`-element list, to get the `n/2 + i` indexed element, a singly linked list with head/tail references must traverse `n/2 + i` nodes, because it cannot "go back" from tail. A doubly linked list with head/tail references only has to traverse `n/2 - i` nodes, because it can "go back" from tail, traversing the list in reverse order.

Example Code in C

A basic SinglyLinkedList example in Java

A basic implementation for singly-linked-list in java - that can add integer values to the end of the list, remove the first value encountered value from list, return an array of values at a given instant and determine whether a given value is present in the list.

Node.java

```
package com.example.so.ds;


/**
 * <p> Basic node implementation </p>
 * @since 20161008
 * @author Ravindra HV
 * @version 0.1
 */
public class Node {

    private Node next;
    private int value;

    public Node(int value) {
```

```
        this.value=value;
    }

    public Node getNext() {
        return next;
    }

    public void setNext(Node next) {
        this.next = next;
    }

    public int getValue() {
        return value;
    }


}
```

SinglyLinkedList.java

```
package com.example.so.ds;


/**
 * <p> Basic single-linked-list implementation </p>
 * @since 20161008
 * @author Ravindra HV
 * @version 0.2
 */
public class SinglyLinkedList {

    private Node head;
    private volatile int size;

    public int getSize() {
        return size;
    }

    public synchronized void append(int value) {

        Node entry = new Node(value);
        if(head == null) {
            head=entry;
        }
        else {
            Node temp=head;
            while( temp.getNext() != null) {
                temp=temp.getNext();
            }
            temp.setNext(entry);
        }

        size++;
    }


    public synchronized boolean removeFirst(int value) {
        boolean result = false;

        if( head == null ) { // or size is zero..
```

```
            // no action
        }
        else if( head.getValue() == value ) {
            head = head.getNext();
            result = true;
        }
        else {

            Node previous = head;
            Node temp = head.getNext();
            while( (temp != null) && (temp.getValue() != value) ) {
                previous = temp;
                temp = temp.getNext();
            }

            if((temp != null) && (temp.getValue() == value)) { // if temp is null then not
found..
                previous.setNext( temp.getNext() );
                result = true;
            }

        }

        if(result) {
            size--;
        }

        return result;

    }

    public synchronized int[] snapshot() {
        Node temp=head;
        int[] result = new int[size];
        for(int i=0;i<size;i++) {
            result[i]=temp.getValue();
            temp = temp.getNext();
        }
        return result;
    }

    public synchronized boolean contains(int value) {
        boolean result = false;
        Node temp = head;

        while(temp!=null) {
            if(temp.getValue() == value) {
                result=true;
                break;
            }
            temp=temp.getNext();
        }
        return result;
    }

}
```

TestSinglyLinkedList.java

```
package com.example.so.ds;
```

```
import java.util.Arrays;
import java.util.Random;


/**
 *
 * <p> Test-case for single-linked list</p>
 * @since 20161008
 * @author Ravindra HV
 * @version 0.2
 *
 */
public class TestSinglyLinkedList {

    /**
     * @param args
     */
    public static void main(String[] args) {

        SinglyLinkedList singleLinkedList = new SinglyLinkedList();

        int loop = 11;
        Random random = new Random();

        for(int i=0;i<loop;i++) {

            int value = random.nextInt(loop);
            singleLinkedList.append(value);

            System.out.println();
            System.out.println("Append :"+value);
            System.out.println(Arrays.toString(singleLinkedList.snapshot()));
            System.out.println(singleLinkedList.getSize());
            System.out.println();
        }

        for(int i=0;i<loop;i++) {
            int value = random.nextInt(loop);
            boolean contains = singleLinkedList.contains(value);
            singleLinkedList.removeFirst(value);

            System.out.println();
            System.out.println("Contains :"+contains);
            System.out.println("RemoveFirst :"+value);
            System.out.println(Arrays.toString(singleLinkedList.snapshot()));
            System.out.println(singleLinkedList.getSize());
            System.out.println();
        }


    }

}
```

Read Linked List online: https://riptutorial.com/data-structures/topic/2967/linked-list

Chapter 7: Matrices

Remarks

This article aims to explain what a matrix is and how to use it.

Examples

Intro to matrices

Matrices are essentially two-dimensional arrays.

That means that it associates (i, j) coordinates, where i is the row and j is the column, to a value.

So, you could have :

$m_{3,\ 4}$ = "Hello"

The easiest implementation is to make an array or arrays. In python, that would go as follows.

```
matrix = [["a", "b", "c"], ["d", "e", "f"], ["g", "h", "i"]]
print(matrix[1][2]) #returns "f"
```

Read Matrices online: https://riptutorial.com/data-structures/topic/7455/matrices

Chapter 8: Queue

Examples

Intro to Queue

The queue is a FIFO (first-in, first-out) data-structure, i.e. the first element added to the queue will be the first element removed ("first out").

Let us consider the example of customers waiting to be helped. Alice, Bob, and Dan are all at the supermarket. Alice is ready to pay, so she approaches the cashier. Alice is now in the queue. She is the only person in the queue, so she is both at the front and at the back.

Now, the queue looks like this:

```
               |-------|
               | Alice | cashier
               |-------|
                 ▲   ▲
back of queue ──┘   └── front of queue
```

Now Bob and then Dan approach the cashier. They are added to the queue. Alice is still at the front, and Dan is at the back:

```
               |-------||-------||-------|
               |  Dan  ||  Bob  || Alice | cashier
               |-------||-------||-------|
                 ▲                   ▲
back of queue ──┘                   └── front of queue
```

Adding a person to the queue is the enqueue operation. Alice, Bob, and Dan have been enqueued. As the cashier helps each customer, they will be removed from the queue. This is the dequeue operation. Customers, which represent the data elements in our queue, are dequeued from the front of the queue. This means that the first customer that approached up to the cashier was the first customer to be helped (FIFO).

Queue Implementation by using array.

Queue follow FIFO as it is mentioned in the introduction. Five major operations:

1. Enqueue(x): pushes x to the back of the queue.
2. Dequeue(): pops an element from the front of the queue.
3. isEmpty(): Finds whether the queue is empty or not.
4. isFull(): Finds whether the queue is full or not.
5. frontValue(): Returns the front value of the Queue.

All the operations are constant O(1) time.

Code:

```
#include<stdio.h>

#define MAX 4

int front = -1;
int rear = -1;
int a[MAX];

bool isFull() {
  if(rear == MAX-1)
    return true;
  else
    return false;
}

bool isEmpty() {
  if(rear == -1 && front==-1)
    return true;
  else
    return false;
}

void enqueue(int data) {
  if (isFull()){
    printf("Queue is full\n");
    return;
  } else if(isEmpty()) {
    front = rear =0;
  } else {
    rear = rear + 1;
    a[rear] = data;
  }
}

void deque(){
  if(isEmpty()){
    printf("Queue is empty\n");
    return;
  } else if(front == rear) {
    front =-1;
    rear =-1;
  } else {
    front = front + 1;
  }
}

void print(){
  printf("Elements in Queue are\n");
  for(int i = front;i<=rear;i++){
    printf("%d ",a[i]);
  }
  printf("\n");
}

int frontValue(){
  printf("The front element after set of enqueue and dequeue is %d\n", a[front]);
}

int main(){
```

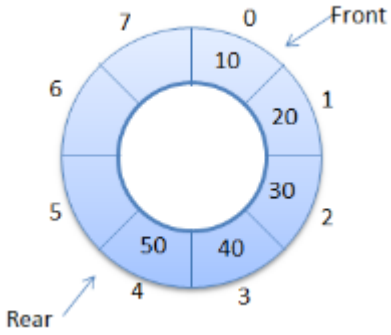
```
  deque();     // Queue is empty message will be thrown
  enqueue(10);
  print();
  enqueue(20);
  print();
  enqueue(30);
  print();
  enqueue(40);
  frontValue();
  print();
  enqueue(50);
  frontValue();
  deque();
  deque();
  enqueue(50);
  frontValue();
  print();
  return 0;
}
```

Implementation of a circular Queue

Memory is efficiently organized in a circular queue as compared to linear queue.

In Linear Queue:

In Circular Queue:



Remaining spaces can be used:

Code for it to do the same:

```
#include<stdio.h>
#define MAX 10000
int front = -1;
int rear = -1;
int a[MAX];
```

```
bool isFull() {
  if((rear+1) % MAX == front)
    return true;
  else
    return false;
}

bool isEmpty() {
  if(rear == -1 && front==-1)
    return true;
  else
    return false;
}

void enqueue(int data) {
  if (isFull()){
    printf("Queue is full\n");
    return;
  } else if(isEmpty()) {
    front = rear = 0;
  } else {
    rear = (rear+1) % MAX;
    a[rear] = data;
  }
}

void deque() {
  if(isEmpty()){
    printf("Queue is empty\n");
    return;
  } else if(front == rear) {
    front =-1;
    rear =-1;
  } else {
    front = (front+1) % MAX;
  }
}

int frontValue() {
  return(a[front]);
}

int main() {
  deque();
  enqueue(10);
  enqueue(20);
  enqueue(30);
  enqueue(40);
  enqueue(50);
  frontValue();
  return 0;
}
```

All operations have O(1) time complexity.

Linked List representation of Queue

Linked list representation is more efficient in terms of memory managerment.

Code to show enqueue and deque in a Queue using Linklist in O(1) time.

```
#include<stdio.h>
#include<malloc.h>

struct node{
    int data;
    node* next;
};

node* front = NULL;
node* rear = NULL;

void enqueue(int data){      //adds element to end

    struct node* temp = (struct node*)malloc(sizeof(struct node*));
    temp->data = data;
    temp->next =NULL;

    if(front== NULL && rear == NULL){
        front = rear = temp;
        return;
    }
    rear->next = temp;
    rear= temp;
}

void deque(){     //removes element from front
    node* temp = front;

    if(front== NULL && rear == NULL){
        return;
    }
    else if (front==rear){
        front =rear = NULL;
    }
    else
    front= front ->next;

    free(temp);
}

void print(){
    node* temp = front;

    for(; temp != rear; temp=temp->next){
        printf("%d ",temp->data);
    }
    //for printing the rear element

        printf("%d ",temp->data);
    printf("\n");
}



int main(){

    enqueue(20);
    enqueue(50);
        enqueue(70);
```

```
        printf("After set of enques\n");
        print();

        deque();
        printf("After 1 deque\n");
        print();

        return 0;


}
```

Read Queue online: https://riptutorial.com/data-structures/topic/7097/queue

Chapter 9: Segment Tree

Examples

Introduction To Segment Tree

Suppose we have an array:

```
+-------+-----+-----+-----+-----+-----+-----+
| Index |  0  |  1  |  2  |  3  |  4  |  5  |
+-------+-----+-----+-----+-----+-----+-----+
| Value |  -1 |  3  |  4  |  0  |  2  |  1  |
+-------+-----+-----+-----+-----+-----+-----+
```

We want to perform some query on this array. For example:

- What is the minimum from index-**2** to index-**4**? -> 0
- What is the maximum from index-**0** to index-**3**? -> 4
- What is the summation from index-**1** to index-**5**? -> 10

How do we find it out?

Brute Force:
We could traverse the array from the starting index to the finishing index and answer our query. In this approach, each query takes $O(n)$ time where **n** is the difference between the starting index and finishing index. But what if there are millions of numbers and millions of queries? For **m** queries, it would take $O(mn)$ time. So for 10^5 values in our array, if we conduct 10^5 queries, for worst case, we'll need to traverse 10^{10} items.

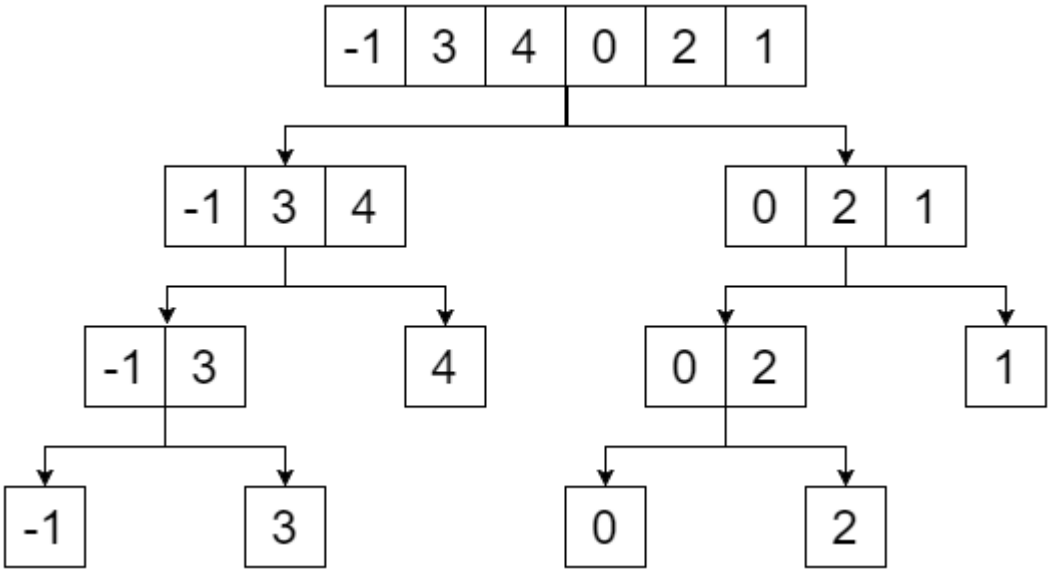
Dynamic Programming:
We can create a 6X6 matrix to store the values for different ranges. For range minimum query(RMQ), our array would look like:

```
          0     1     2     3     4     5
    +-----+-----+-----+-----+-----+-----+-----+
    |     |  -1 |  3  |  4  |  0  |  2  |  1  |
    +-----+-----+-----+-----+-----+-----+-----+
0   |  -1 |  -1 |  -1 |  -1 |  -1 |  -1 |  -1 |
    +-----+-----+-----+-----+-----+-----+-----+
1   |  3  |     |  3  |  3  |  0  |  0  |  0  |
    +-----+-----+-----+-----+-----+-----+-----+
2   |  4  |     |     |  4  |  0  |  0  |  0  |
    +-----+-----+-----+-----+-----+-----+-----+
3   |  0  |     |     |     |  0  |  0  |  0  |
    +-----+-----+-----+-----+-----+-----+-----+
4   |  2  |     |     |     |     |  2  |  1  |
    +-----+-----+-----+-----+-----+-----+-----+
5   |  1  |     |     |     |     |     |  1  |
    +-----+-----+-----+-----+-----+-----+-----+
```

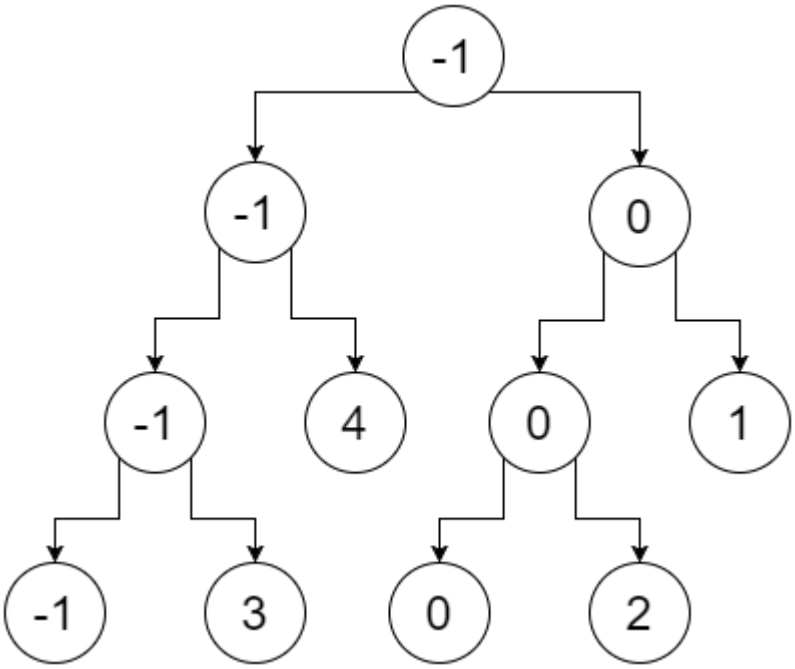
Once we have this matrix build, it would be sufficient to answer all the RMQs. Here, **Matrix[i][j]** would store the minimum value from index-**i** to index-**j**. For example: The minimum from index-**2** to index-**4** is **Matrix[2][4]** = **0**. This matrix answers the query in $O(1)$ time, but it takes **O(n²)** time to build this matrix and $O(n^2)$ space to store it. So if **n** is a really big number, this doesn't scale very well.

Segment Tree:
A segment tree is a tree data structure for storing intervals, or segments. It allows querying which of the stored segments contain a given point. It takes $O(n)$ time to build a segment tree, it takes $O(n)$ space to maintain it and it answers a query in $O(logn)$ time.

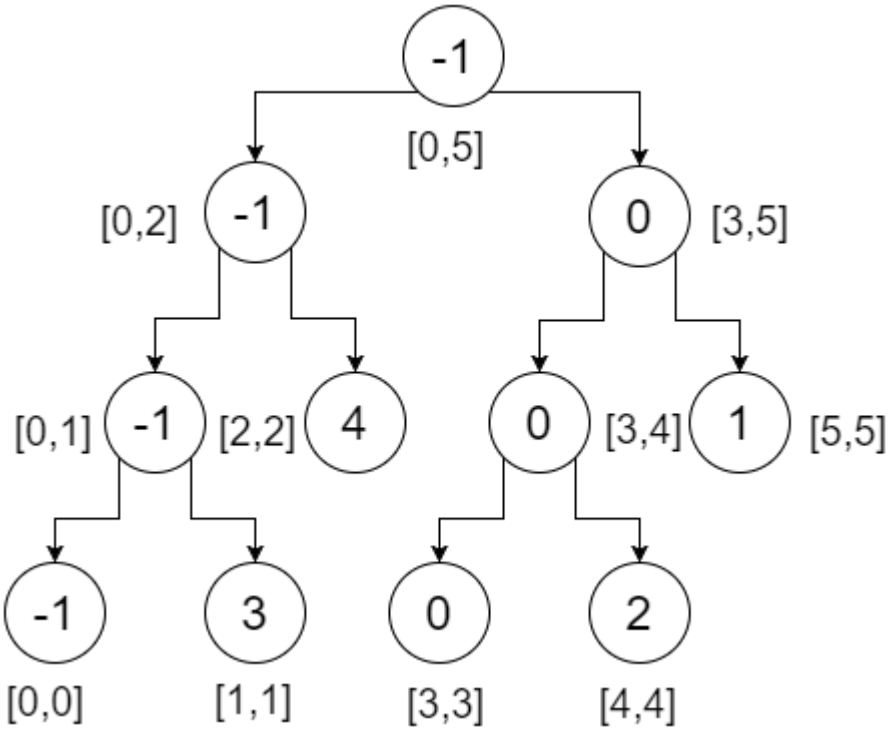
Segment tree is a binary tree and the elements of the given array will be the leaves of that tree. We'll create the segment tree by dividing the array in half till we reach a single element. So our division would provide us with:



Now if we replace the non-leaf nodes with the minimum value of their leaves, we get:



So this is our segment tree. We can notice that, the root node contains the minimum value of the whole array(range [0,5]), its left child contains the minimum value of our left array(range [0,2]), right child contains the minimum value of our right array(range [3,5]) and so on. The leaf nodes contain minimum value of each individual points. We get:



Now let's do a range query on this tree. To do a range query, we need to check three conditions:

- Partial Overlap: We check both leaves.
- Total Overlap: We return the value stored in the node.
- No Overlap: We return a very large value or infinity.

Let's say, we want to check range **[2,4]**, that means we want to find the minimum from index-**2** to **4** . We'll start from the root and check if the range in our nodes is overlapped by our query range or not. Here,

- **[2,4]** doesn't completely overlap **[0,5]**. So we'll check both directions.
  - At left subtree, **[2,4]** partially overlaps **[0,2]**. We'll check both directions.
    - At left subtree, **[2,4]** does not overlap **[0,1]**, so this is not going to contribute to our answer. We return **infinity**.
    - At right subtree, **[2,4]** totally overlaps **[2,2]**. We return **4**.
      The minimum of these two returned values(4, infinity) is **4**. We get **4** from this portion.
  - At right subtree, **[2,4]** partially overlaps. So we'll check both directions.
    - At left subtree, **[2,4]** completely overlaps **[3,4]**. We return **0**.
    - At right subtree, **[2,4]** doesn't overlap **[5,5]**. We return **infinity**.
      The minimum of these two returned values(0, infinity) is **0**. We get **0** from this portion.
- The minimum of the returned values(4,0) is **0**. This is our desired minimum in range **[2,4]**.

Now we can check that, no matter what our query is, it would take maximum **4** steps to find the

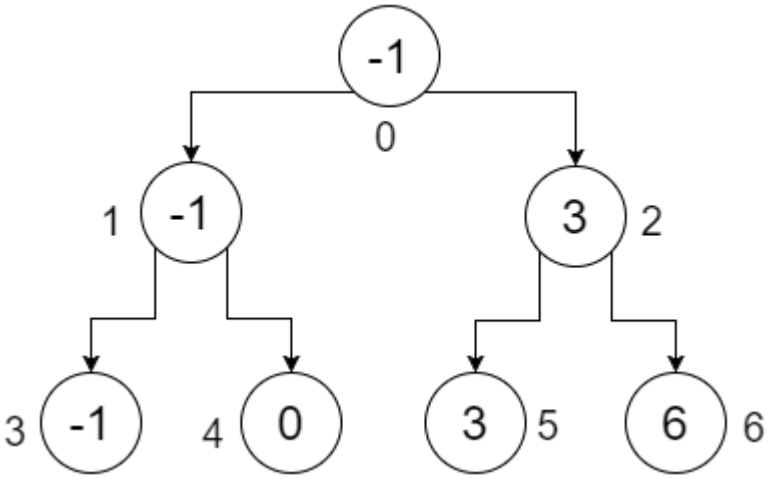
desired value from this segment tree.

Use:

- Range Minimum Query
- Lowest Common Ancestor
- Lazy Propagation
- Dynamic Subtree Sum
- Neglect & Min
- Dual Fields
- Finding k-th Smallest Element
- Finding Number of Unordered Pairs

Implementation of Segment Tree Using Array

Let's say, we have an array: `Item = {-1, 0, 3, 6}`. We want to construct **SegmentTree** array to find out the minimum value in a given range. Our segment tree will look like:



The numbers below the nodes show the indices of each values that we'll store in our **SegmentTree** array. We can see that, to store 4 elements, we needed an array of size 7. This value is determined by:

```
Procedure DetermineArraySize(Item):
multiplier := 1
n := Item.size
while multiplier < n
    multiplier := multiplier * 2
end while
size := (2 * multiplier) - 1
Return size
```

So if we had an array of length 5, the size of our **SegmentTree** array would be: (**8** * **2**) - **1** = **15**. Now, to determine the position of left and right child of a node, if a node is in index **i**, then the position of its:

- Left Child is denoted by: **(2** * **i)** + **1**.
- Right Child is denoted by: **(2** * **i)** + **2**.

And the index of the **parent** of any **node** in index **i** can be determined by: **(i** - **1)/2**.

So the **SegmentTree** array representing our example would look like:

```
   0     1     2     3     4     5     6
+-----+-----+-----+-----+-----+-----+-----+
|  -1 |  -1 |  3  |  -1 |  0  |  3  |  6  |
+-----+-----+-----+-----+-----+-----+-----+
```

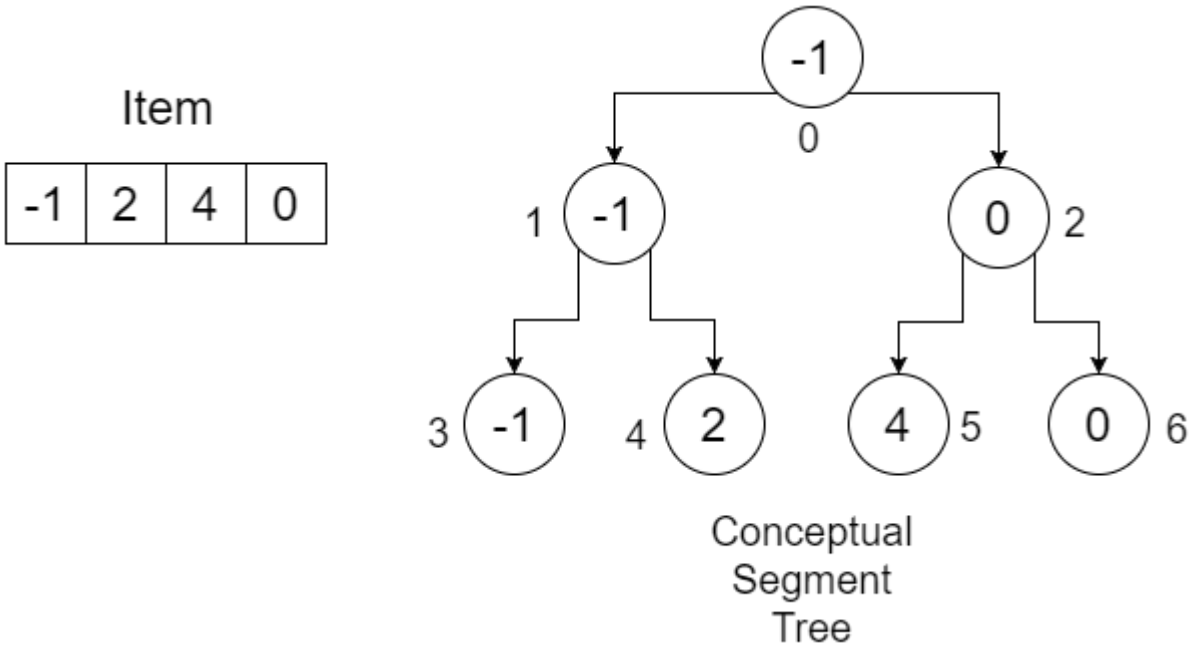
Let's look at the pseudo-code to construct this array:

```
Procedure ConstructTree(Item, SegmentTree, low, high, position):
if low is equal to high
    SegmentTree[position] := Item[low]
else
    mid := (low+high)/2
    constructTree(Item, SegmentTree, low, mid, 2*position+1)
    constructTree(Item, SegmentTree, mid+1, high, 2*position+2)
    SegmentTree[position] := min(SegmentTree[2*position+1], SegmentTree[2*position+2])
end if
```

At first, we take input of the values and initialize the **SegmentTree** array with `infinity` using the length of the **Item** array as its size. We call the the procedure using:

- low = Starting index of **Item** array.
- high = Finishing index of **Item** array.
- position = 0, indicates the **root** of our Segment Tree.

Now, let's try to understand the procedure using an example:



The size of our **Item** array is **4**. We create an array of length **(4** * **2)** - **1** = **7** and initialize them with `infinity`. You can use a very large value for it. Our array would look like:

```
   0     1     2     3     4     5     6
+-----+-----+-----+-----+-----+-----+-----+
```

```
| inf | inf | inf | inf | inf | inf | inf |
+-----+-----+-----+-----+-----+-----+-----+
```

Since this is a recursive procedure, we'll see the operation of the `ConstructTree` using a recursion table that keeps track of `low`, `high`, `position`, `mid` and `calling line` at each call. At first, we call **ConstructTree(Item, SegmentTree, 0, 3, 0)**. Here, `low` is not same as `high`, we'll get a `mid`. The `calling line` indicates which `ConstructTree` is called after this statement. We denote the `ConstructTree` calls inside the procedure as **1** and **2** respectively. Our table will look like:

```
+-----+------+----------+-----+--------------+
| low | high | position | mid | calling line |
+-----+------+----------+-----+--------------+
|  0  |   3  |     0    |  1  |       1      |
+-----+------+----------+-----+--------------+
```

So when we call `ConstructTree-1`, we pass: `low = 0`, `high = mid = 1`, `position = 2*position+1 = 2*0+1 = 1`. One thing you can notice, that is `2*position+1` is the left child of **root**, which is **1**. Since `low` is not equal to `high`, we get a `mid`. Our table will look like:

```
+-----+------+----------+-----+--------------+
| low | high | position | mid | calling line |
+-----+------+----------+-----+--------------+
|  0  |   3  |     0    |  1  |       1      |
+-----+------+----------+-----+--------------+
|  0  |   1  |     1    |  0  |       1      |
+-----+------+----------+-----+--------------+
```

In the next recursive call, we pass `low = 0`, `high = mid = 0`, `position = 2*position+1 = 2*1+1=3`. Again the left child of index **1** is **3**. Here, `low` is e`high`, so we set `SegmentTree[position] = SegmentTree[3] = Item[low] = Item[0] = -1`. Our **SegmentTree** array will look like:

```
   0     1     2     3     4     5     6
+-----+-----+-----+-----+-----+-----+-----+
| inf | inf | inf |  -1 | inf | inf | inf |
+-----+-----+-----+-----+-----+-----+-----+
```

Our recursion table will look like:

```
+-----+------+----------+-----+--------------+
| low | high | position | mid | calling line |
+-----+------+----------+-----+--------------+
|  0  |   3  |     0    |  1  |       1      |
+-----+------+----------+-----+--------------+
|  0  |   1  |     1    |  0  |       1      |
+-----+------+----------+-----+--------------+
|  0  |   0  |     3    |     |              |
+-----+------+----------+-----+--------------+
```

So you can see, **-1** has got its correct position. Since this recursive call is complete, we go back to the previous row of our recursion table. The table:

```
+-----+------+----------+-----+--------------+
```

```
| low | high | position | mid | calling line |
+-----+------+----------+-----+--------------+
|  0  |   3  |     0    |  1  |       1      |
+-----+------+----------+-----+--------------+
|  0  |   1  |     1    |  0  |       1      |
+-----+------+----------+-----+--------------+
```

In our procedure, we execute the `ConstructTree-2` call. This time, we pass `low = mid+1 = 1`, `high = 1`, `position = 2*position+2 = 2*1+2 = 4`. Our `calling line` changes to **2**. We get:

```
+-----+------+----------+-----+--------------+
| low | high | position | mid | calling line |
+-----+------+----------+-----+--------------+
|  0  |   3  |     0    |  1  |       1      |
+-----+------+----------+-----+--------------+
|  0  |   1  |     1    |  0  |       2      |
+-----+------+----------+-----+--------------+
```

Since, `low` is equal to `high`, we set: `SegmentTree[position] = SegmentTree[4] = Item[low] = Item[1] = 2`. Our **SegmentTree** array:

```
   0     1     2     3     4     5     6
+-----+-----+-----+-----+-----+-----+-----+
| inf | inf | inf |  -1 |  2  | inf | inf |
+-----+-----+-----+-----+-----+-----+-----+
```

Our recursion table:

```
+-----+------+----------+-----+--------------+
| low | high | position | mid | calling line |
+-----+------+----------+-----+--------------+
|  0  |   3  |     0    |  1  |       1      |
+-----+------+----------+-----+--------------+
|  0  |   1  |     1    |  0  |       2      |
+-----+------+----------+-----+--------------+
|  1  |   1  |     4    |     |              |
+-----+------+----------+-----+--------------+
```

Again you can see, **2** has got its correct position. After this recursive call, we go back to the previous row of our recursion table. We get:

```
+-----+------+----------+-----+--------------+
| low | high | position | mid | calling line |
+-----+------+----------+-----+--------------+
|  0  |   3  |     0    |  1  |       1      |
+-----+------+----------+-----+--------------+
|  0  |   1  |     1    |  0  |       2      |
+-----+------+----------+-----+--------------+
```

We execute the last line of our procedure, `SegmentTree[position] = SegmentTree[1] = min(SegmentTree[2*position+1], SegmentTree[2*position+2]) = min(SegmentTree[3], SegmentTree[4]) = min(-1,2) = -1`. Our **SegmentTree** array:

```
   0     1     2     3     4     5     6
+-----+-----+-----+-----+-----+-----+-----+
| inf |  -1 | inf |  -1 |  2  | inf | inf |
+-----+-----+-----+-----+-----+-----+-----+
```

Since this recursion call is complete, we go back to the previous row of our recursion table and call `ConstructTree-2`:

```
+-----+------+----------+-----+--------------+
| low | high | position | mid | calling line |
+-----+------+----------+-----+--------------+
|  0  |   3  |     0    |  1  |       2      |
+-----+------+----------+-----+--------------+
```

We can see that the left portion of our segment tree is complete. If we continue in this manner, after completing the whole procedure we'll finally get a completed **SegmentTree** array, that'll look like:

```
   0     1     2     3     4     5     6
+-----+-----+-----+-----+-----+-----+-----+
|  -1 |  -1 |  0  |  -1 |  2  |  4  |  0  |
+-----+-----+-----+-----+-----+-----+-----+
```

The time and space complexity of constructing this **SegmentTree** array is: `O(n)`, where **n** denotes the number of elements in **Item** array. Our constructed **SegmentTree** array can be used to perform *Range Minimum Query(RMQ)*. To construct an array to perform *Range Maximum Query*, we need to replace the line:

```
SegmentTree[position] := min(SegmentTree[2*position+1], SegmentTree[2*position+2])
```

with:

```
SegmentTree[position] := max(SegmentTree[2*position+1], SegmentTree[2*position+2])
```

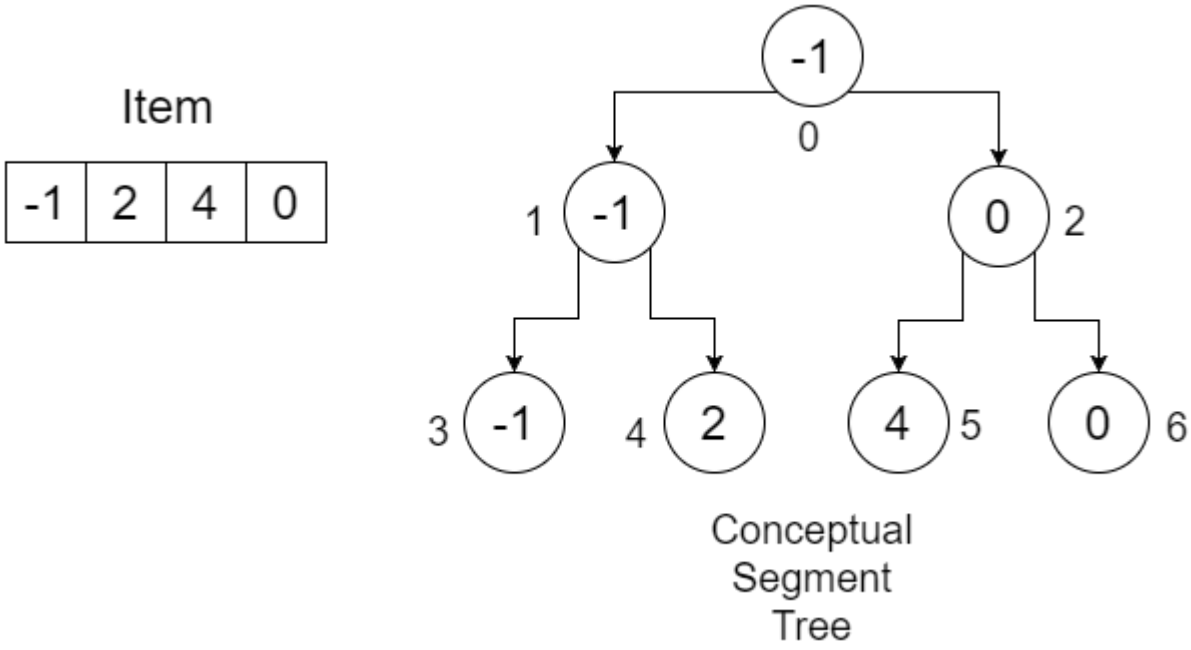
Performing a Range Minimum Query

The procedure to perform a RMQ is already shown in introduction. The pseudo-code for checking Range Minimum Query will be:

```
Procedure RangeMinimumQuery(SegmentTree, qLow, qHigh, low, high, position):
if qLow <= low and qHigh >= high        //Total Overlap
    Return SegmentTree[position]
else if qLow > high || qHigh < low      //No Overlap
    Return infinity
else                                    //Partial Overlap
    mid := (low+high)/2
    Return min(RangeMinimumQuery(SegmentTree, qLow, qHigh, low, mid, 2*position+1),
               RangeMinimumQuery(SegmentTree, qLow, qHigh, mid+1, high, 2*position+2))
end if
```

Here, `qLow = The lower range of our query`, `qHigh = The upper range of our query`. `low = starting`

`index of Item array`, `high = Finishing index of Item array`, `position = root = 0`. Now let's try to understand the procedure using the example we created before:



Our **SegmentTree** array:

```
   0     1     2     3     4     5     6
+-----+-----+-----+-----+-----+-----+-----+
|  -1 |  -1 |  0  |  -1 |  2  |  4  |  0  |
+-----+-----+-----+-----+-----+-----+-----+
```

We want to find the minimum in range **[1,3]**.
Since this is a recursive procedure, we'll see the operation of the `RangeMinimumQuery` using a recursion table that keeps track of `qLow`, `qHigh`, `low`, `high`, `position`, `mid` and `calling line`. At first, we call **RangeMinimumQuery(SegmentTree, 1, 3, 0, 3, 0**. Here, the first two conditions are not met(partial overlap). We'll get a `mid`. The `calling line` indicates which `RangeMinimumQuery` is called after this statement. We denote the `RangeMinimumQuery` calls inside the procedure as **1** and **2** respectively. Our table will look like:

```
+------+-------+-----+------+----------+-----+--------------+
| qLow | qHigh | low | high | position | mid | calling line |
+------+-------+-----+------+----------+-----+--------------+
|   1  |   3   |  0  |   3  |     0    |  1  |       1      |
+------+-------+-----+------+----------+-----+--------------+
```

So when we call `RangeMinimumQuery-1`, we pass: `low = 0`, `high = mid = 1`, `position = 2*position+1 = 1`. One thing you can notice, that is `2*position+1` is the left child of a **node**. That means we're checking the left child of **root**. Since **[1,3]** partially overlaps **[0,1]**, the first two conditions are not met, we get a `mid`. Our table:

```
+------+-------+-----+------+----------+-----+--------------+
| qLow | qHigh | low | high | position | mid | calling line |
+------+-------+-----+------+----------+-----+--------------+
|   1  |   3   |  0  |   3  |     0    |  1  |       1      |
```

```
+------+-------+-----+------+----------+-----+--------------+
|   1  |   3   |  0  |   1  |     1    |  0  |       1      |
+------+-------+-----+------+----------+-----+--------------+
```

In the next recursive call, we pass `low = 0`, `high = mid = 0`, `position = 2*position+1 = 3`. We reach the leftmost leaf of our tree. Since **[1,3]** doesn't overlap with **[0,0]**, we return `infinity` to our calling function. Recursion table:

```
+------+-------+-----+------+----------+-----+--------------+
| qLow | qHigh | low | high | position | mid | calling line |
+------+-------+-----+------+----------+-----+--------------+
|   1  |   3   |  0  |   3  |     0    |  1  |       1      |
+------+-------+-----+------+----------+-----+--------------+
|   1  |   3   |  0  |   1  |     1    |  0  |       1      |
+------+-------+-----+------+----------+-----+--------------+
|   1  |   3   |  0  |   0  |     3    |     |              |
+------+-------+-----+------+----------+-----+--------------+
```

Since this recursive call is complete, we go back to the previous row of our recursion table. We get:

```
+------+-------+-----+------+----------+-----+--------------+
| qLow | qHigh | low | high | position | mid | calling line |
+------+-------+-----+------+----------+-----+--------------+
|   1  |   3   |  0  |   3  |     0    |  1  |       1      |
+------+-------+-----+------+----------+-----+--------------+
|   1  |   3   |  0  |   1  |     1    |  0  |       1      |
+------+-------+-----+------+----------+-----+--------------+
```

In our procedure, we execute `RangeMinimumQuery-2` call. This time, we pass `low = mid+1 = 1`, `high = 1` and `position = 2*position+2 = 4`. Our `calling line changes to **2**`. We get:

```
+------+-------+-----+------+----------+-----+--------------+
| qLow | qHigh | low | high | position | mid | calling line |
+------+-------+-----+------+----------+-----+--------------+
|   1  |   3   |  0  |   3  |     0    |  1  |       1      |
+------+-------+-----+------+----------+-----+--------------+
|   1  |   3   |  0  |   1  |     1    |  0  |       2      |
+------+-------+-----+------+----------+-----+--------------+
|   1  |   3   |  1  |   1  |     4    |     |              |
+------+-------+-----+------+----------+-----+--------------+
```

So we are going to the right child of previous node. This time there is a total overlap. We return the value `SegmentTree[position] = SegmentTree[4] = 2`.

```
+------+-------+-----+------+----------+-----+--------------+
| qLow | qHigh | low | high | position | mid | calling line |
+------+-------+-----+------+----------+-----+--------------+
|   1  |   3   |  0  |   3  |     0    |  1  |       1      |
+------+-------+-----+------+----------+-----+--------------+
```

Back at the calling function, we are checking what is the minimum of the two returned values of two calling functions. Obviously the minimum is **2**, we return **2** to the calling function. Our recursion

table looks like:

```
+------+-------+-----+------+----------+-----+--------------+
| qLow | qHigh | low | high | position | mid | calling line |
+------+-------+-----+------+----------+-----+--------------+
|   1  |   3   |  0  |   3  |     0    |  1  |       1      |
+------+-------+-----+------+----------+-----+--------------+
```

We're done looking at the left portion of our segment tree. Now we'll call `RangeMinimumQuery-2` from here. We'll pass `low = mid+1 = 1+1 =2`, `high = 3` and `position = 2*position+2 = 2`. Our table:

```
+------+-------+-----+------+----------+-----+--------------+
| qLow | qHigh | low | high | position | mid | calling line |
+------+-------+-----+------+----------+-----+--------------+
|   1  |   3   |  0  |   3  |     0    |  1  |       1      |
+------+-------+-----+------+----------+-----+--------------+
|   1  |   3   |  2  |   3  |     2    |     |              |
+------+-------+-----+------+----------+-----+--------------+
```

There is a total overlap. So we return the value: `SegmentTree[position] = SegmentTree[2] = 0`. We come back to the recursion from where these two children were called and get the minimum of **(4,0)**, that is **0** and return the value.

After executing the procedure, we get **0**, which is the minimum from index-**1** to index-**3**.

The runtime complexity for this procedure is `O(logn)` where **n** is the number of elements in the **Items** array. To perform a Range Maximum Query, we need to replace the line:

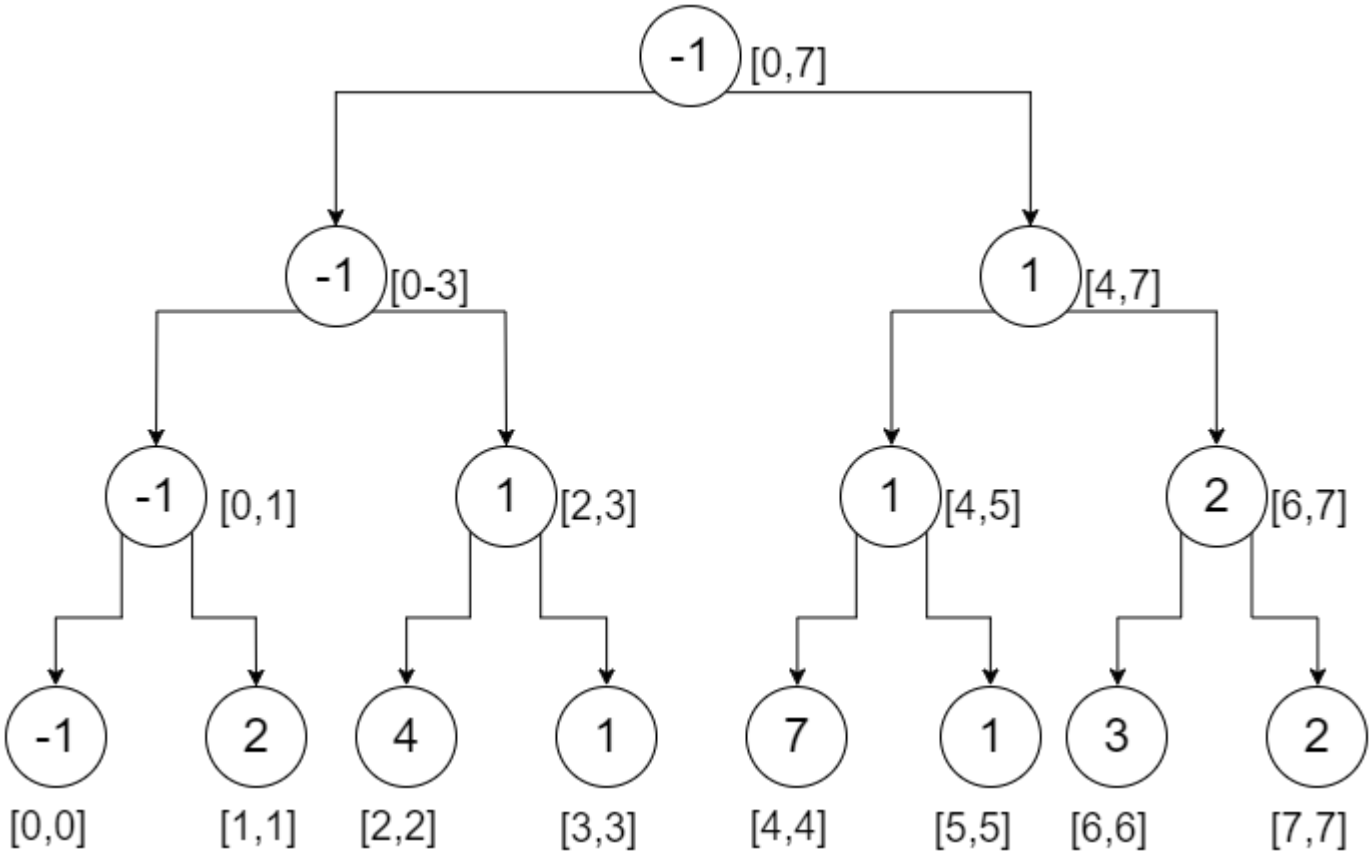
```
Return min(RangeMinimumQuery(SegmentTree, qLow, qHigh, low, mid, 2*position+1),
               RangeMinimumQuery(SegmentTree, qLow, qHigh, mid+1, high, 2*position+2))
```

with:

```
Return max(RangeMinimumQuery(SegmentTree, qLow, qHigh, low, mid, 2*position+1),
               RangeMinimumQuery(SegmentTree, qLow, qHigh, mid+1, high, 2*position+2))
```

Lazy Propagation

Let's say, you have already created a segment tree. You are required to update the values of the array, this will not only change the leaf nodes of your segment tree, but also the minimum/maximum in some nodes. Let's look at this with an example:



This is our minimum segment tree of **8** elements. To give you a quick reminder, each nodes represent the minimum value of the range mentioned beside them. Let's say, we want to increment the value of the first item of our array by **3**. How can we do that? We'll follow the way in which we conducted RMQ. The process would look like:

- At first, we traverse the root. **[0,0]** partially overlaps with **[0,7]**, we go to both directions.
  - At left subtree, **[0,0]** partially overlaps with **[0,3]**, we go to both directions.
    - At left subtree, **[0,0]** partially overlaps with **[0,1]**, we go to both directions.
      - At left subtree, **[0,0]** totally overlaps with **[0,0]**, and since its the leaf node, we update the node by increasing it s value by **3**. And return the value **-1** + **3** = **2**.
      - At right subtree, **[1,1]** doesn't overlap with **[0,0]**, we return the value at the node(**2**).
        The minimum of these two returned values(**2**, **2**) are **2**, so we update the value of the current node and return it.
    - At right subtree **[2,3]** doesn't overlap with **[0,0]**, we return the value of the node. (**1**).
      Since the minimum of these two returned values (**2**, **1**) is **1**, we update the value of the current node and return it.
  - At right subtree **[4,7]** doesn't overlap with **[0,0]**, we return the value of the node. (**1**).
- Finally the value of the root node is updated since the minimum of the two returned values **(1,1)** is **1**.

We can see that, updating a single node requires `O(logn)` time complexity, where **n** is the number of leaf nodes. So if we are asked to update some nodes from **i** to **j**, we'll require `O(nlogn)` time complexity. This becomes cumbersome for a large value of **n**. Let's be *Lazy* i. e., do work only

when needed. How? When we need to update an interval, we will update a node and mark its child that it needs to be updated and update it when needed. For this we need an array **lazy** of the same size as that of a segment tree. Initially all the elements of the **lazy** array will be **0** representing that there is no pending update. If there is non-zero element in **lazy[i]**, then this element needs to update node **i** in the segment tree before making any query operations. How are we going to do that? Let's look at an example:

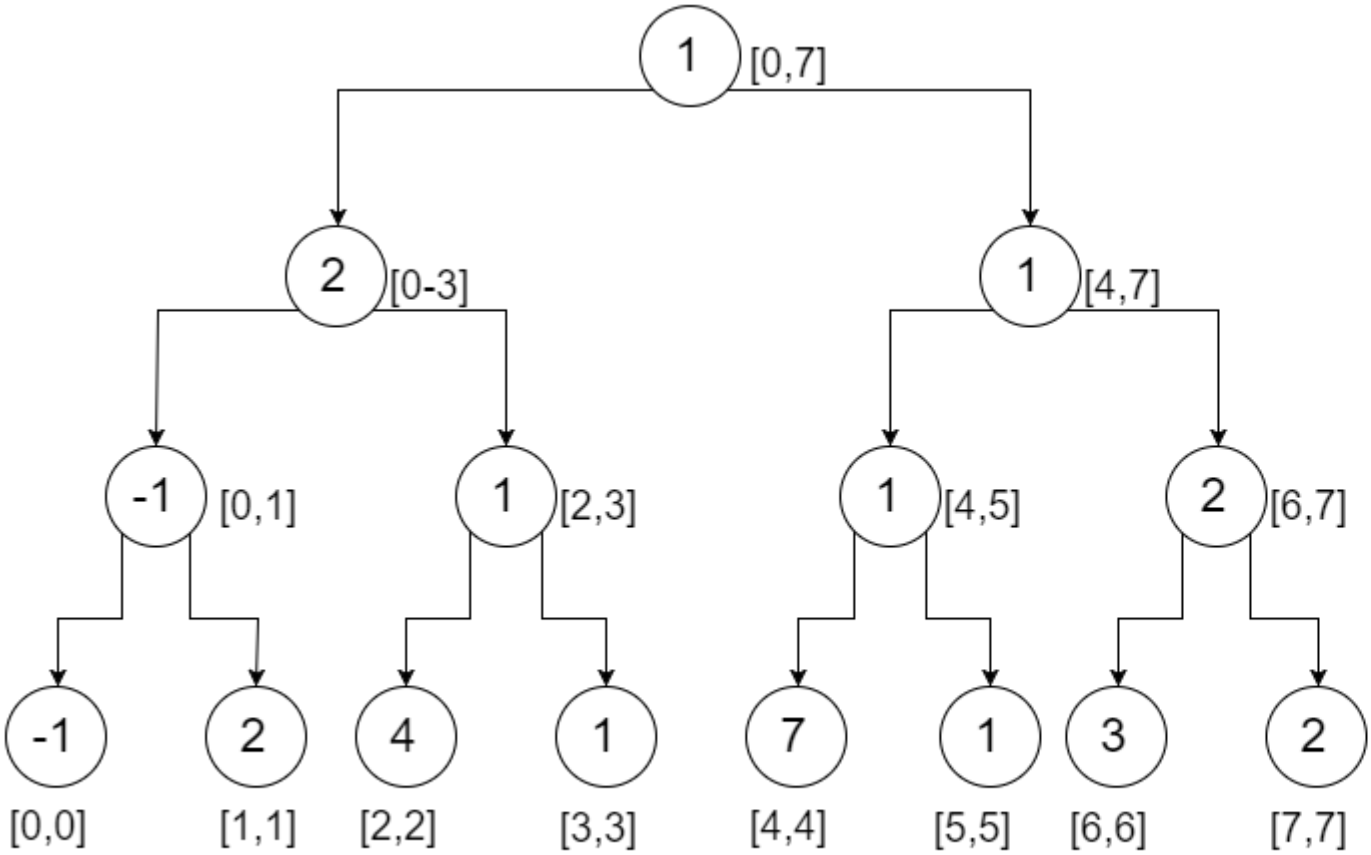
Let's say, for our example tree, we want to execute some queries. These are:

- increment **[0,3]** by **3**.
- increment **[0,3]** by **1**.
- increment **[0,0]** by **2**.

increment [0,3] by 3:

- We start from the root node. At first, we check if this value is up-to-date. For this we check our **lazy** array which is **0**, that means the value is up-to-date. **[0,3]** partially overlaps **[0,7]**. So we go to both the directions.
    - At the left subtree, there's no pending update. **[0,3]** totally overlaps **[0,3]**. So we update the value of the node by **3**. So the value becomes **-1** + **3** = **2**. This time, we're not going to go all the way. Instead of going down, we update the corresponding child in the lazy tree of our current node and increment them by **3**. We also return the value of the current node.
    - At the right subtree, there's no pending update. **[0,3]** doesn't overlap **[4,7]**. So we return the value of the current node **(1)**.
      The minimum of two returned values (**2**, **1**) is **1**. We update the root node to **1**.

Our Segment Tree and Lazy Tree would look like:



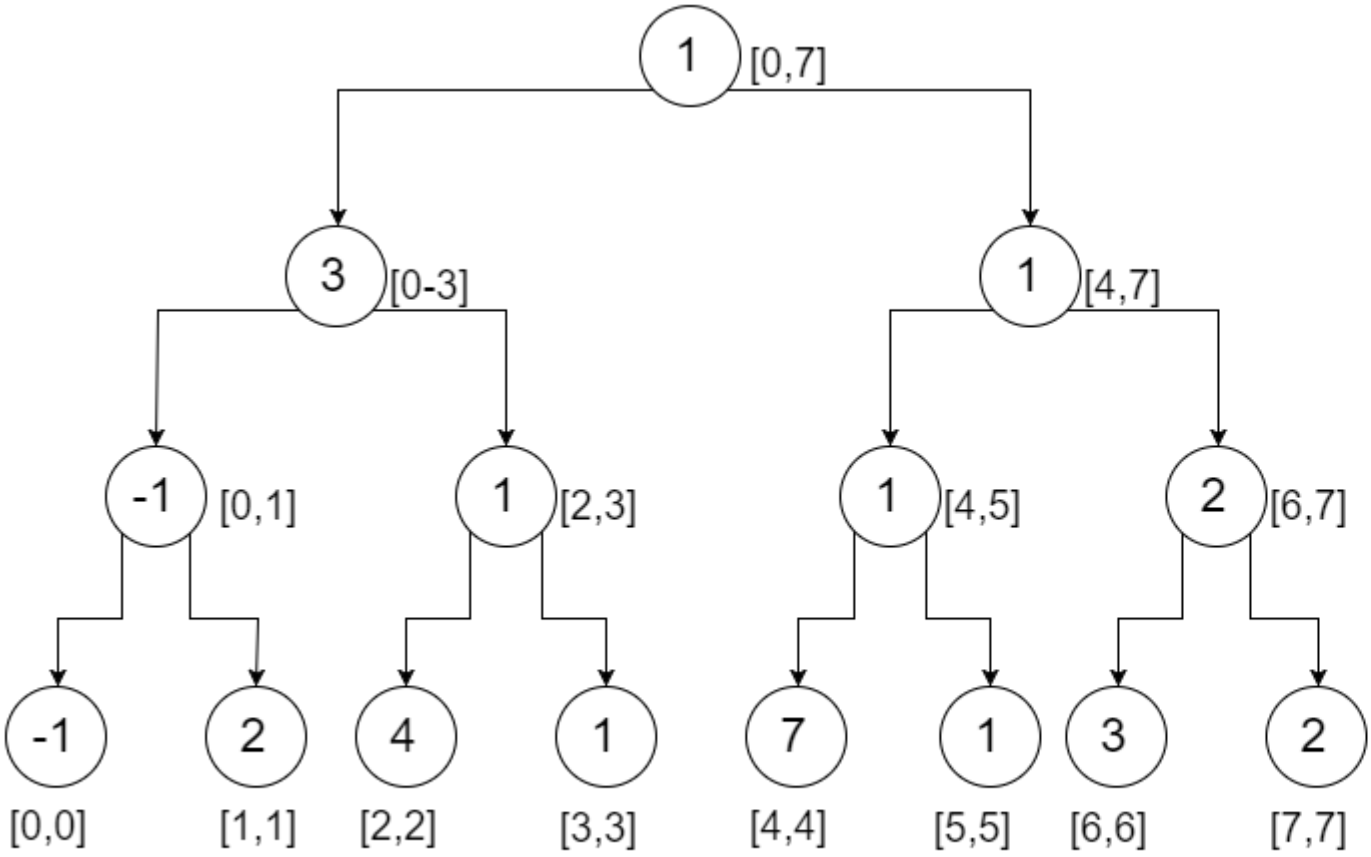
Segment Tree

The non-zero values in nodes of our Lazy Tree represents, there are updates pending in those nodes and below. We'll update them if required. If we are asked, what is the minimum in range **[0,3]**, we'll come to the left subtree of the root node and since there's no pending updates, we'll return **2**, which is correct. So this process doesn't affect the correctness of our segment tree algorithm.

increment [0,3] by 1:

- We start from the root node. There's no pending update. **[0,3]** partially overlaps **[0,7]**. So we go to both directions.
  - In the left subtree, there's no pending update. **[0,3]** completely overlaps **[0,3]**. We update the current node: **2** + **1** = **3**. Since this is an internal node, we update its children in the Lazy Tree to be incremented by **1**. We'll also return the value of the current node (**3**).
  - In the right subtree, there's no pending update. **[0,3]** doesn't overlap **[4,7]**. We return the value of the current node (**1**).
- We update the root node by taking the minimum of two returned values(**3**, **1**).

Our Segment Tree and Lazy Tree will look like:



Segment Tree

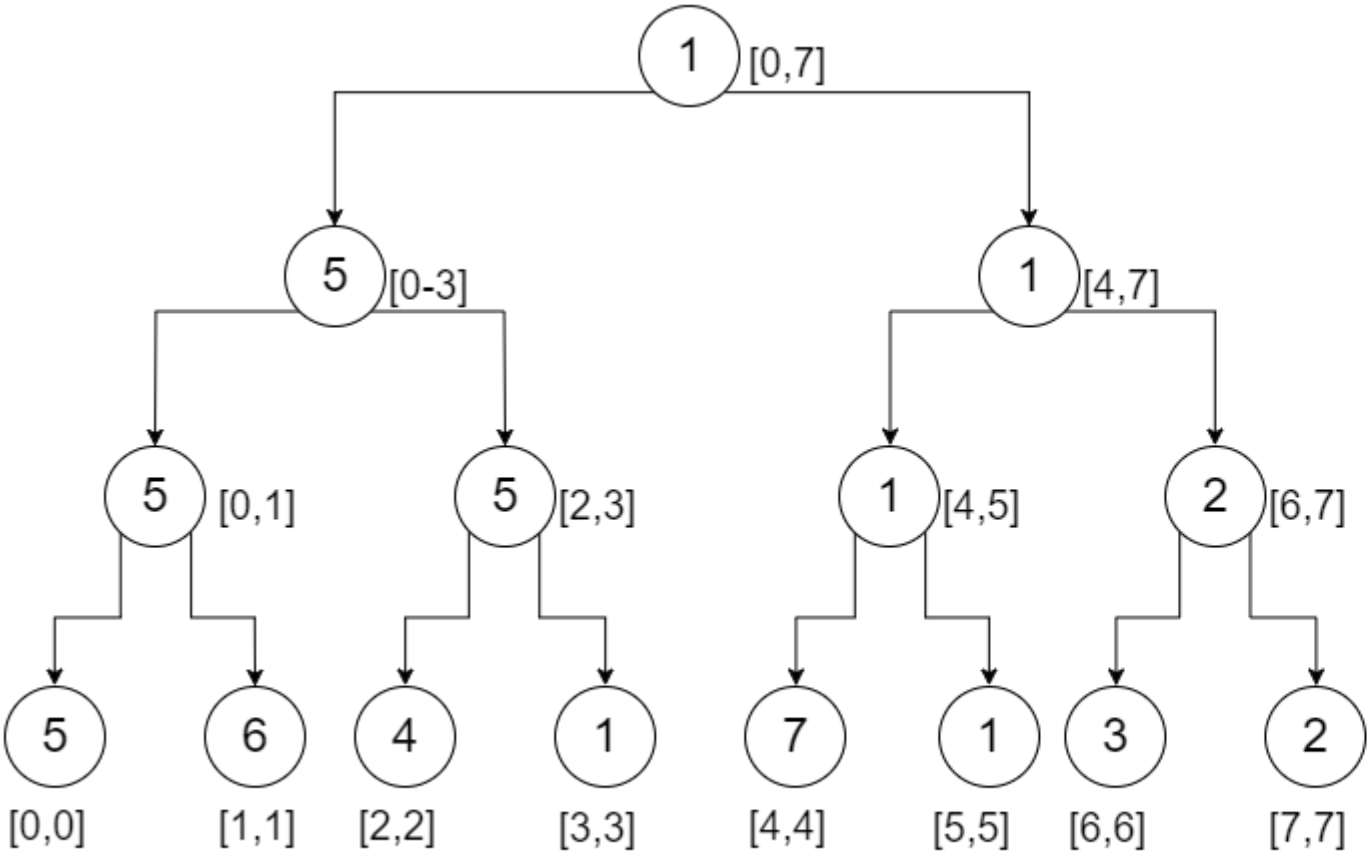
As you can see, we're accumulating the updates at Lazy Tree but not pushing it down. This is what Lazy Propagation means. If we hadn't used it, we had to push the values down to the leaves, which would cost us more unnecessary time complexity.

increment [0,0] by 2:

- We start from the root node. Since root is up-to-date and **[0,0]** partially overlaps **[0,7]**, we go to both directions.
  - At the left subtree, the current node is up-to-date and **[0,0]** partially overlaps **[0,3]**, we go to both directions.
    - At the left subtree, the current node in Lazy Tree has a non-zero value. So there is an update which has not been propagated yet to this node. We're going to first update this node in our Segment Tree. So this becomes **-1** + **4** = **3**. Then we're going to propagate this **4** to its children in the Lazy Tree. As we have already updated the current node, we'll change the value of current node in Lazy Tree to **0**. Then **[0,0]** partially overlaps **[0,1]**, so we go to both directions.
      - At the left node, the value needs to be updated since there is a non-zero value in the current node of Lazy Tree. So we update the value to **-1** + **4** = **3** . Now, since **[0,0]** totally overlaps **[0,0]**, we update the value of the current node to **3** + **2** = **5**. This is a leaf node, so we need not to propagate the value anymore. We update the corresponding node at the Lazy Tree to **0** since all the values have been propagated up till this node. We return the value of the current node(**5**).
      - At the right node, the value needs to be updated. So the value becomes: **4**

+ **2** = **6**. Since **[0,0]** doesn't overlap **[1,1]**, we return the value of the current node(**6**). We also update the value in Lazy Tree to **0**. No propagation is needed since this is a leaf node.
We update the current node using the minimum of two returned values(**5**,**6**). We return the value of the current node(**5**).
    - At the right subtree, there's a pending update. We update the value of the node to **1** + **4** = **5**. Since this is not a leaf node, we propagate the value to its children in our Lazy Tree and update the current node to **0**. Since **[0,0]** doesn't overlap with **[2,3]**, we return the value of our current node(**5**).
    We update the current node using the minimum of the returned values(**5**, **5**) and return the value(**5**).
  - At the right subtree, there's no pending update and since **[0,0]** doesn't overlap **[4,7]**, we return the value of the current node(**1**).
- We update the root node using the minimum of the two returned values(**5**,**1**).

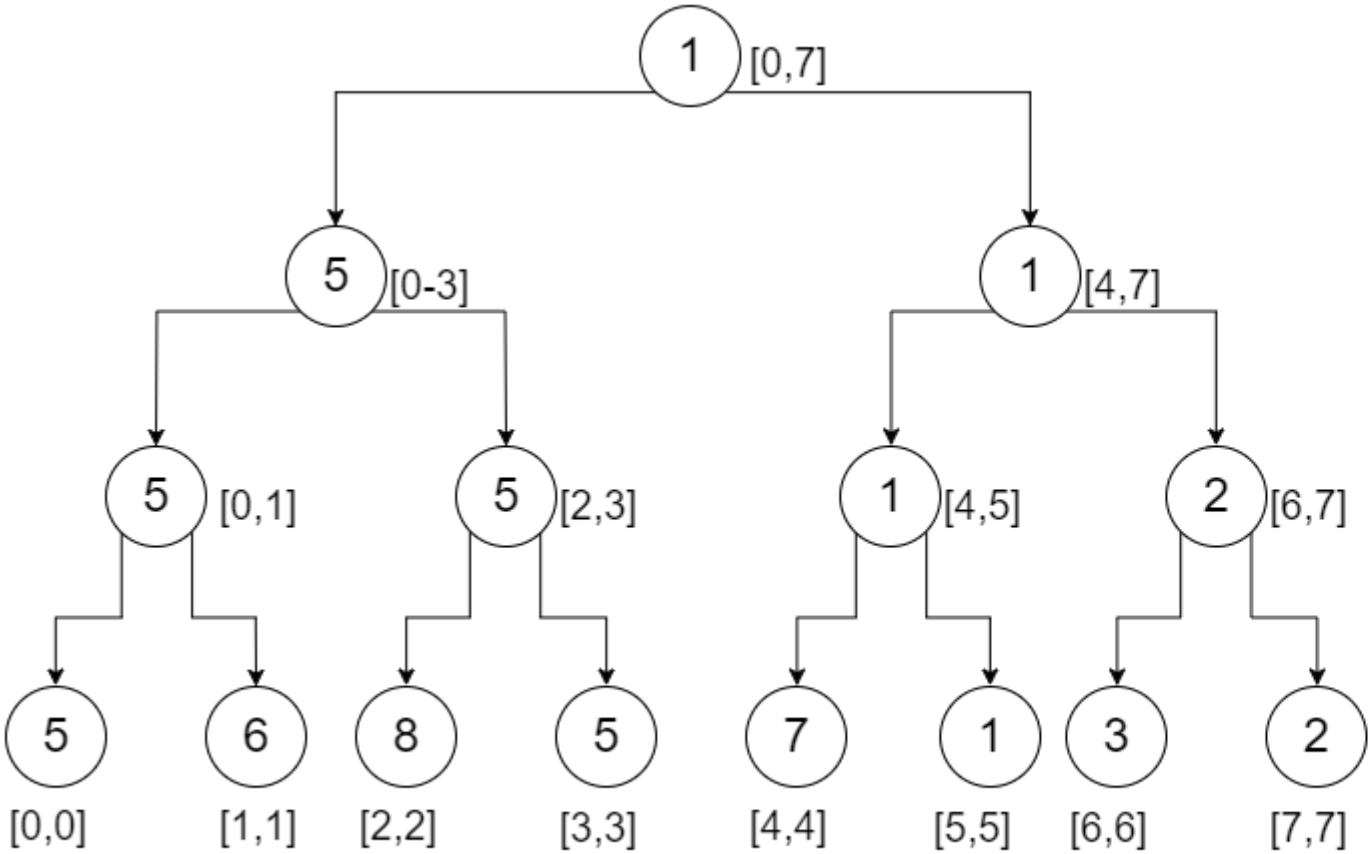
Our Segment Tree and Lazy Tree will look like:



Segment Tree

We can notice that, the value at **[0,0]**, when needed, got all the increment.

Now if you are asked to find the minimum in range **[3,5]**, if you have understood up to this point, you can easily figure out how the query would go and the returned value will be **1**. Our segment Tree and Lazy Tree would look like:



Segment Tree

We have simply followed the same process we followed in finding RMQ with added constraints of checking the Lazy Tree for pending updates.

Another thing we can do is instead of returning values from each node, since we know what will be the child node of our current node, we can simply check them to find the minimum of these two.

The pseudo-code for updating in Lazy Propagation would be:

```
Procedure UpdateSegmentTreeLazy(segmentTree, LazyTree, startRange,
                                endRange, delta, low, high, position):
if low > high                                                   //out of bounds
    Return
end if
if LazyTree[position] is not equal to 0                         //update needed
    segmentTree[position] := segmentTree[position] + LazyTree[position]
    if low is not equal to high                                 //non-leaf node
        LazyTree[2 * position + 1] := LazyTree[2 * position + 1] + delta
        LazyTree[2 * position + 2] := LazyTree[2 * position + 2] + delta
    end if
    LazyTree[position] := 0
end if
if startRange > low or endRange < high                          //doesn't overlap
    Return
end if
if startRange <= low && endRange >= high                        //total overlap
    segmentTree[position] := segmentTree[position] + delta
    if low is not equal to high
        LazyTree[2 * position + 1] := LazyTree[2 * position + 1] + delta
```

```
        LazyTree[2 * position + 2] := LazyTree[2 * position + 2] + delta
    end if
    Return
end if
//if we reach this portion, this means there's a partial overlap
mid := (low + high) / 2
UpdateSegmentTreeLazy(segmentTree, LazyTree, startRange,
                                endRange, delta, low, mid, 2 * position + 1)
UpdateSegmentTreeLazy(segmentTree, LazyTree, startRange,
                                endRange, delta, mid + 1, high, 2 * position + 2)
segmentTree[position] := min(segmentTree[2 * position + 1],
                                                segmentTree[2 * position + 2])
```

And the pseudo-code for RMQ in Lazy Propagation will be:

```
Procedure RangeMinimumQueryLazy(segmentTree, LazyTree, qLow, qHigh, low, high, position):
if low > high
    Return infinity
end if
if LazyTree[position] is not equal to 0                      //update needed
    segmentTree[position] := segmentTree[position] + LazyTree[position]
    if low is not equal to high
        segmentTree[position] := segmentTree[position] + LazyTree[position]
    if low is not equal to high                              //non-leaf node
        LazyTree[2 * position + 1] := LazyTree[2 * position + 1] + LazyTree[position]
        LazyTree[2 * position + 2] := LazyTree[2 * position + 2] + LazyTree[position]
    end if
    LazyTree[position] := 0
end if
if qLow > high and qHigh < low                                //no overlap
    Return infinity
end if
if qLow <= low and qHigh >= high                              //total overlap
    Return segmentTree[position]
end if
//if we reach this portion, this means there's a partial overlap
mid := (low + high) / 2
Return min(RangeMinimumQueryLazy(segmentTree, LazyTree, qLow, qHigh,
                                        low, mid, 2 * position + 1),
           RangeMinimumQueryLazy(segmentTree, LazyTree, qLow, qHigh,
                                        mid + 1, high, 2 * position + 1)
```

Read Segment Tree online: https://riptutorial.com/data-structures/topic/7908/segment-tree

Chapter 10: Stack

Examples

Intro to Stack

The stack is a LIFO (last-in, first-out) data-structure, i.e. the most recent (or "last in") element added to the stack will be the first element removed ("first out").

Let us consider the example of books in a box. Only one book can be added or removed from from the box at a time, and it can only be added and removed from the top.

Now, the box with the first two books looks like this:

```
|--------|
| book 2 | <------ top of box
|--------|
| book 1 | <------ bottom of box
|--------|
```

If we add book 3, it will be at the top. The rest of the books, which were added before book 3, will remain below it:

```
|--------|
| book 3 | <------ top of box
|--------|
| book 2 |
|--------|
| book 1 | <------ bottom of box
|--------|
```

The box is like a stack and the books are data elements. Adding a book to the box is the push operation while removing/getting a book from the top of the box is the pop operation. If we perform the pop operation, we will get book 3, and the box will go back to the way it was before we pushed book 3. This means that the last (or most recent) element that we put in was the first element we got out (LIFO). In order to get book 1, we have to perform two more pops: one to remove book 2, and the second to get book 1.

Implementation of the stack using an array. For this, we need an index pointing to the top location (tos). Every time an element is pushed into the stack the tos increments by one and whenever a pop operation is performed the index pointing the top of the stack (tos) is decremented by one.

PUSH:

Before the PUSH operation

```
                              tos
                               |
                               |
```

```
        |-----|-----|-----|-----|-----|
        | i1  | i2  | i3  |  i4 |     |
        |-----|-----|-----|-----|-----|
```

```
void push(dataElment item){
    stack[top]=item; //item = i4
    top++;

}
```

After the PUSH operation The stack has a pointer to the top of the stack. Whenever the push operation is called it places the value at the top and updates it the value.

```
        tos -- Top of the stack
                            tos
                             |
                             |
        |-----|-----|-----|-----|
        | i1  | i2  | i3  |     |
        |-----|-----|-----|-----|
```

POP : The pop operation removes the content from the top of the stack and updates the value of tos by decrementing by 1

Before the pop operation:

```
                                tos
                                 |
                                 |
        |-----|-----|-----|-----|-----|
        | i1  | i2  | i3  |  i4 |     |
        |-----|-----|-----|-----|-----|
```

```
dataElment pop(){
    dataElment value = stack[tos--];
    return value;
}
```

After the pop operation:

```
                            tos
                             |
                             |
        |-----|-----|-----|-----|
        | i1  | i2  | i3  |  i4 |
        |-----|-----|-----|-----|
        When a push operation is performed it overwrites i4.
```

Using stacks to find palindromes

A palindrome is a word that can be read both ways, like 'kayak'.

This example will show how to find if a given word is a palindrome using Python3.

First, we need to turn a string into a stack (we will use arrays as stacks).

```
str = "string"
stk = [c for c in str] #this makes an array: ["s", "t", "r", "i", "n", "g"]
stk.append("s") #adds a letter to the array
stk.pop() #pops the last element
```

Now, we have to invert the word.

```
def invert(str):
    stk = [c for c in str]
    stk2 = []
    while len(stk) > 0:
        stk2.append(stk.pop())
    #now, let's turn stk2 into a string
    str2 = ""
    for c in stk2:
        str2 += c
    return str2
```

Now we can compare the word and the inverted form.

```
def palindrome(str):
    return str == invert(str)
```

Stack Implementation by using array and Linked List

Array Implementation

```
#include<stdio.h>

#define MAX 100000                    //Global variables
int top = -1;
int a[MAX];

void push(int x){

    if(top == MAX-1){                         // Check whether stack is full
        printf("Stack Overflow\n");
        return;
    }

    a[++top] = x;                             // Otherwise increment top and insert x
}

void pop(){
    if(top == -1){                                      // if top = -1, empty stack
        printf("Empty stack\n");
        return;
    }
    top--;
}
```

```
void print(){                           //printing stack values

    for(int i=0;i <= top;i++){
        printf("%d ",a[i]);
    }
    printf("\n");
}

void topValue(){                          // Method can print the top value of a stack
    printf("%d",a[top]);
}
int main(){

    push(5);
    push(20);
    push(15);
    print();
    pop();
    print();
    push(35);
    print();
    topValue();

}
```

Linked List implementation

```
#include<stdio.h>
#include<malloc.h>

 struct node{
    int data;
    node* link;
};
node* top = NULL;

void push(int data){

    node* temp = (struct node*)malloc(sizeof(struct node*));
    temp->data = data;
    temp->link = top;
    top = temp;
}

void pop(){

    if(top == NULL) return;
    node* temp = top;
    top = top->link;
    free(temp);
}
void print(){

    node* t = top;
    while(t != NULL){
        printf("%d ",t->data);
        t=t->link;
    }
    printf("\n");
}
```

```
void topValue(){
    printf("%d ",top->data);
}
int main(){

    push(5);
    push(20);
    push(15);
    print();
    pop();
    print();
    push(35);
    print();
    topValue();
}
```

Checking Balanced Parentheses

A bracket is considered to be any one of the following characters: `(`, `)`, `{`, `}`, `[`, or `]`.

Two brackets are considered to be a matched pair if the an opening bracket (i.e., `(`, `[`, or `{`) occurs to the left of a closing bracket (i.e., `)`, `]`, or `}`) of the exact same type. There are three types of matched pairs of brackets: `[]`, `{}`, and `()`.

A matching pair of brackets is not balanced if the set of brackets it encloses are not matched. For example, `{[(])}` is not balanced because the contents in between `{` and `}` are not balanced. The pair of square brackets encloses a single, unbalanced opening bracket, `(`, and the pair of parentheses encloses a single, unbalanced closing square bracket, `]`.

By this logic, we say a sequence of brackets is considered to be balanced if the following conditions are met:

> It contains no unmatched brackets.
>
> The subset of brackets enclosed within the confines of a matched pair of brackets is also a matched pair of brackets.

Algorithm:

1. Declare a stack (say `stack`).
2. Now traverse the string input.

- a) If the current character is a starting bracket (i.e. `(` or `{` or `[`) then push it to stack.
- b) If the current character is a closing bracket (i.e. `)` or `}` or `]`) then pop from stack. If the popped character is the matching starting bracket then fine else parenthesis are `not balanced` .
- c) If the current character is a closing bracket (i.e. `)` or `}` or `]`) and the stack is empty, then parenthesis are `not balanced`.

3. After complete traversal, if there is some starting bracket left in stack then the string is `not balanced` else we have a `balanced` string.

Read Stack online: https://riptutorial.com/data-structures/topic/6979/stack

Chapter 11: Trie (Prefix Tree/Radix Tree)

Examples

Introduction To Trie

Have you ever wondered how the search engines work? How does Google line-up millions of results in front of you in just a few milliseconds? How does a huge database situated thousands of miles away from you find out information you're searching for and send them back to you? The reason behind this is not possible only by using faster internet and super-computers. Some mesmerizing searching algorithms and data-structures work behind it. One of them is Trie.

Trie, also called *digital tree* and sometimes *radix tree* or *prefix tree* (as they can be searched by prefixes), is a kind of search tree—an ordered tree data structure that is used to store a dynamic set or associative array where the keys are usually strings. It is one of those data-structures that can be easily implemented. Let's say you have a huge database of millions of words. You can use trie to store these information and the complexity of searching these words depends only on the length of the word that we are searching for. That means it doesn't depend on how big our database is. Isn't that amazing?

Let's assume we have a dictionary with these words:

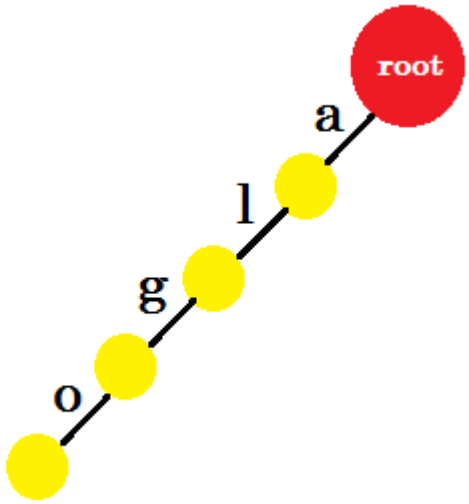
```
algo
algea
also
tom
to
```

We want to store this dictionary in memory in such a way that we can easily find out the word that we're looking for. One of the methods would involve sorting the words lexicographically-how the real-life dictionaries store words. Then we can find the word by doing a *binary search*. Another way is using **Prefix Tree** or **Trie**, in short. The word **'Trie'** comes from the word Re**trie**val. Here, **prefix** denotes *the prefix of string* which can be defined like this: All the substrings that can be created starting from the beginning of a string are called prefix. For example: 'c', 'ca', 'cat' all are the prefix of the string 'cat'.

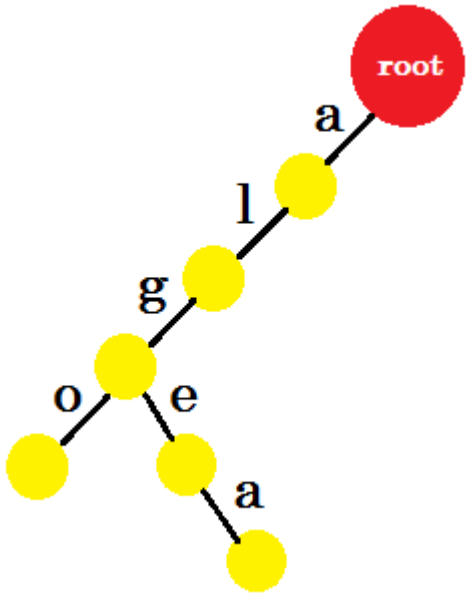
Now let's get back to **Trie**. As the name suggests, we'll create a tree. At first, we have an empty tree with just the root:

We'll insert the word **'algo'**. We'll create a new node and name the edge between these two nodes **'a'**. From the new node we'll create another edge named **'l'** and connect it with another node. In this way, we'll create two new edges for **'g'** and **'o'**. Notice that, we're not storing any information in

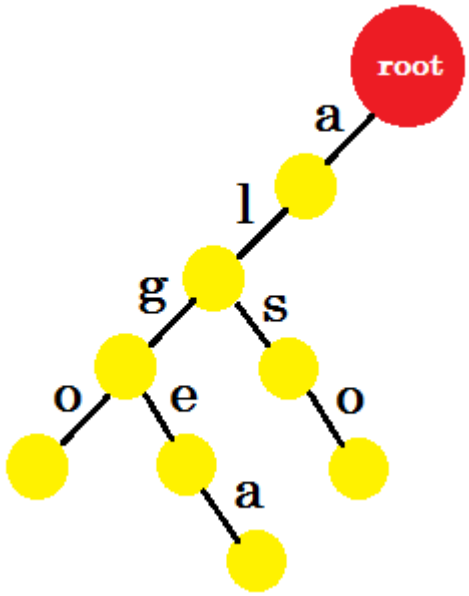
the nodes. For now, we'll only consider creating new edges from these nodes. From now on, let's call an edge named **'x'** - **edge-x**



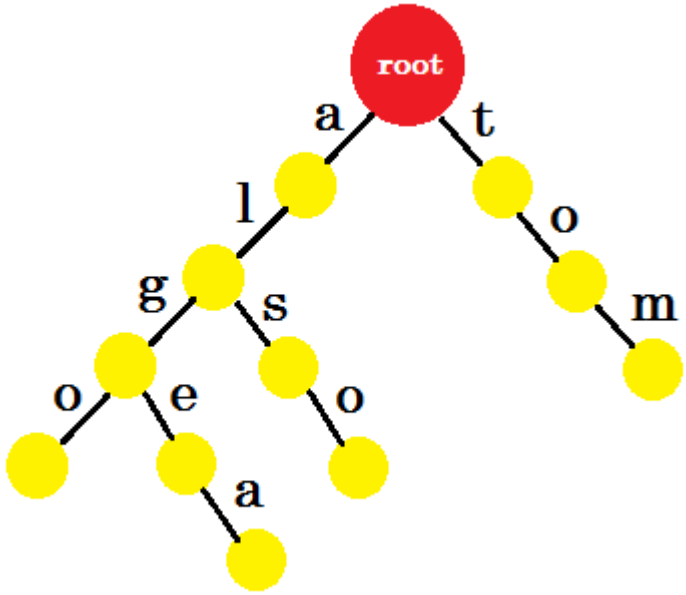
Now we want to add the word **'algea'**. We need an **edge-a** from root, which we already have. So we don't need to add new edge. Similarly, we have an edge from **'a'** to **'l'** and **'l'** to **'g'**. That means **'alg'** is already in the **trie**. We'll only add **edge-e** and **edge-a** with it.



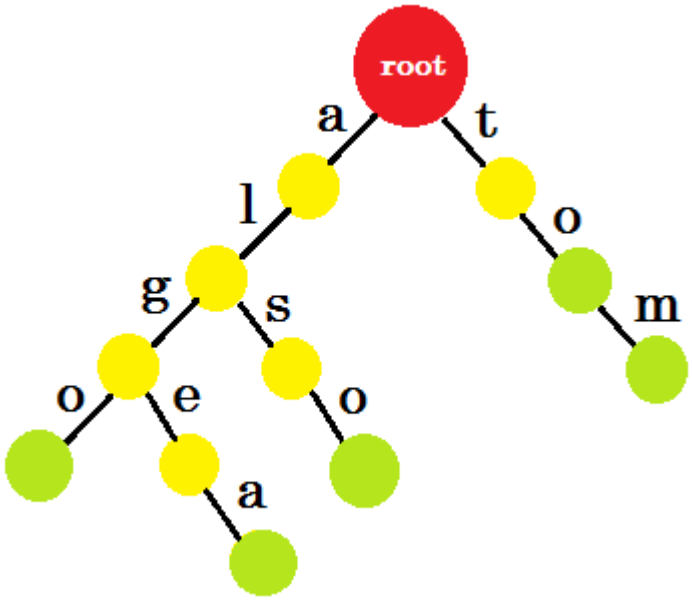
We'll add the word **'also'**. We have the prefix **'al'** from root. We'll only add **'so'** with it.



Let's add **'tom'**. This time, we create a new edge from root as we don't have any prefix of tom created before.



Now how should we add **'to'**? **'to'** is completely a prefix of **'tom'**, so we don't need to add any edge. What we can do is, we can put end-marks in some nodes. We'll put end marks in those nodes where at least one word is completed. Green circle denotes the end-mark. The trie will look like:



You can easily understand why we added the end-marks. We can determine the words stored in trie. The characters will be on the edges and nodes will contain the end-marks.

Now you might ask, what's the purpose of storing words like this? Let's say, you are asked to find the word **'alice'** in the dictionary. You'll traverse the trie. You'll check if there is an **edge-a** from root. Then check from **'a'**, if there's an **edge-l**. After that, you won't find any **edge-i**, so you can come to the conclusion that, the word **alice** doesn't exist in the dictionary.

If you're asked to find the word **'alg'** in the dictionary, you'll traverse **root->a**, **a->l**, **l->g**, but you won't find a green node at the end. So the word doesn't exist in the dictionary. If you search for **'tom'**, you'll end up in a green node, which means the word exists in the dictionary.

Complexity:

The maximum amount of steps needed to search for a word in a **trie** is the length of the word that we're looking for. The complexity is **O(length)**. The complexity for insertion is same. The amount of memory needed to implement **trie** depends on the implementation. We'll see an implementation in another example where we can store $\mathbf{10^6}$ characters (not words, letters) in a **trie**.

Use of Trie:

- To insert, delete and search for a word in dictionary.
- To find out if a string is a prefix of another string.
- To find out how many strings has a common prefix.
- Suggestion of contact names in our phones depending on the prefix we enter.
- Finding out 'Longest Common Substring' of two strings.
- Finding out the length of 'Common Prefix' for two words using 'Longest Common Ancestor'

Implementation of Trie

Before reading this example, it is highly recommended that you read Introduction to Trie first.

One of the easiest ways of implementing **Trie** is using linked list.

Node:

The nodes will consist of:

1. Variable for End-Mark.
2. Pointer Array to the next Node.

The End-Mark variable will simply denote whether it is an end-point or not. The pointer array will denote all the possible edges that can be created. For English alphabets, the size of the array will be 26, as maximum 26 edges can be created from a single node. At the very beginning each value of pointer array will be NULL and the end-mark variable will be false.

```
Node:
Boolean Endmark
Node *next[26]
Node()
    endmark = false
    for i from 0 to 25
        next[i] := NULL
    endfor
endNode
```

Every element in **next[]** array points to another node. **next[0]** points to the node sharing **edge-a**, **next[1]** points to the node sharing **edge-b** and so on. We have defined the constructor of Node to initialize Endmark as false and put NULL in all the values of **next[]** pointer array.

To create **Trie**, at first we'll need to instantiate **root**. Then from the **root**, we'll create other nodes to store information.

Insertion:

```
Procedure Insert(S, root):    // s is the string that needs to be inserted in our dictionary
curr := root
for i from 1 to S.length
    id := S[i] - 'a'
    if curr -> next[id] == NULL
        curr -> next[id] = Node()
    curr := curr -> next[id]
endfor
curr -> endmark = true
```

Here we are working with **a-z**. So to convert their ASCII values to **0-25**, we subtract the ASCII value of **'a'** from them. We will check if current node (curr) has an edge for the character at hand. If it does, we move to the next node using that edge, if it doesn't we create a new node. At the end, we change the endmark to true.

Searching:

```
Procedure Search(S, root)      // S is the string we are searching
curr := root
for i from 1 to S.length
    id := S[i] - 'a'
    if curr -> next[id] == NULL
```

```
        Return false
    curr := curr -> id
Return curr -> endmark
```

The process is similar to insert. At the end, we return the **endmark**. So if the **endmark** is true, that means the word exists in the dictionary. If it's false, then the word doesn't exist in the dictionary.

This was our main implementation. Now we can insert any word in **trie** and search for it.

Deletion:

Sometimes we will need to erase the words that will no longer be used to save memory. For this purpose, we need to delete the unused words:

```
Procedure Delete(curr):       //curr represents the current node to be deleted
for i from 0 to 25
    if curr -> next[i] is not equal to NULL
        delete(curr -> next[i])
    delete(curr)               //free the memory the pointer is pointing to
```

This function goes to all the child nodes of **curr**, deletes them and then **curr** is deleted to free the memory. If we call **delete(root)**, it will delete the whole **Trie**.

Trie can be implemented using *Arrays* too.

Read Trie (Prefix Tree/Radix Tree) online: https://riptutorial.com/data-structures/topic/7178/trie--prefix-tree-radix-tree-

Chapter 12: Union-find data structure

Introduction

A union-find (or disjoint-set) data structure is a simple data structure a partition of a number of elements into disjoint sets. Every set has a representative which can be used to distinguish it from the other sets.

It is used in many algorithms, e.g. to compute minimum spanning trees via Kruskal's algorithm, to compute connected components in undirected graphs and many more.

Examples

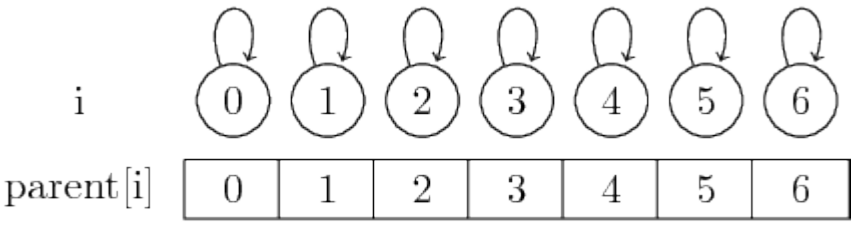
Theory

Union find data structures provide the following operations:

- `make_sets(n)` initializes a union-find data structure with `n` singletons
- `find(i)` returns a representative for the set of element `i`
- `union(i,j)` merges the sets containing `i` and `j`

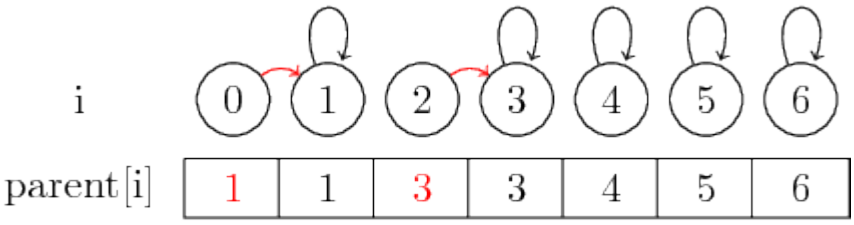
We represent our partition of the elements `0` to `n - 1` by storing a *parent* element `parent[i]` for every element `i` which eventually leads to a *representative* of the set containing `i`.
If an element itself is a representative, it is its own parent, i.e. `parent[i] == i`.

Thus, if we start with singleton sets, every element is its own representative:

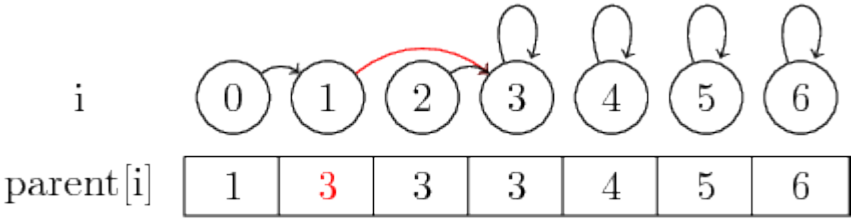


We can find the representative for a given set by simply following these parent pointers.

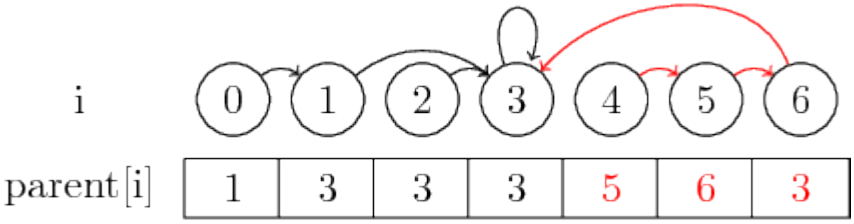
Let us now see how we can merge sets:
If we want to merge the elements 0 and 1 and the elements 2 and 3, we can do this by setting the parent of 0 to 1 and setting the parent of 2 to 3:



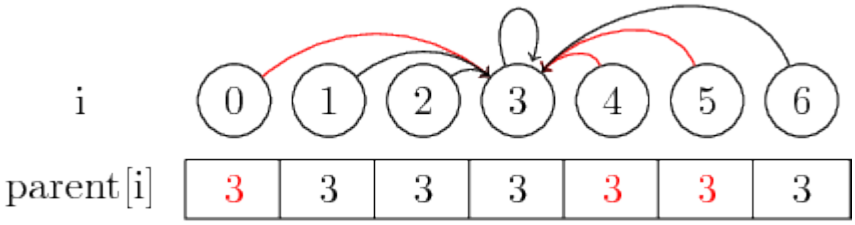
In this simple case, only the elements parent pointer itself must be changed.
If we however want to merge larger sets, we must always change the parent pointer of the *representative* of the set that is to be merged into another set: After **merge(0,3)**, we have set the parent of the representative of the set containing 0 to the representative of the set containing 3



To make the example a bit more interesting, let's now also **merge (4,5), (5,6) and (3,4)**:



The last notion I want to introduce is **path compression**:
If we want to find the representative of a set, we might have to follow several *parent* pointers before reaching the representative. We might make this easier by storing the representative for each set directly in their parent node. We lose the order in which we merged the elements, but can potentially have a large runtime gain. In our case, the only paths that aren't compressed are the



paths from 0, 4 and 5 to 3:

Basic implementation

The most basic implementation of a union-find data structure consists of an array `parent` storing

the a parent element for every element of the structure. Following these parent 'pointers' for an element `i` leads us to the representative `j = find(i)` of the set containing `i`, where `parent[j] = j` holds.

```
using std::size_t;

class union_find {
private:
    std::vector<size_t> parent;  // Parent for every node

public:
    union_find(size_t n) : parent(n) {
        for (size_t i = 0; i < n; ++i)
            parent[i] = i;      // Every element is its own representative
    }

    size_t find(size_t i) const {
        if (parent[i] == i)     // If we already have a representative
            return i;           // return it
        return find(parent[i]); // otherwise return the parent's representative
    }

    void merge(size_t i, size_t j) {
        size_t pi = find(i);
        size_t pj = find(j);
        if (pi != pj) {        // If the elements are not in the same set:
            parent[pi] = pj;   // Join the sets by marking pj as pi's parent
        }
    }
};
```

Improvements: Path compression

If we do many `merge` operations on a union-find data structure, the paths represented by the `parent` pointers might be quite long. *Path compression*, as already described in the theory part, is a simple way of mitigating this issue.

We might try to do path compression on the whole data structure after every *k*-th merge operation or something similar, but such an operation could have a quite large runtime.

Thus, path compression is mostly only used on a small part of the structure, especially the path we walk along to find the representative of a set. This can be done by storing the result of the `find` operation after every recursive subcall:

```
size_t find(size_t i) const {
    if (parent[i] == i)          // If we already have a representative
        return i;                // return it
    parent[i] = find(parent[i]); // path-compress on the way to the representative
    return parent[i];            // and return it
}
```

Improvements: Union by size

In our current implementation of `merge`, we always choose the left set to be the child of the right

set, without taking the size of the sets into consideration. Without this restriction, the paths (without *path compression*) from an element to its representative might be quite long, thus leading to large runtimes on `find` calls.

Another common improvement thus is the *union by size* heuristic, which does exactly what it says: When merging two sets, we always set the larger set to be the parent of the smaller set, thus leading to a path length of at most $\log n$ steps:

We store an additional member `std::vector<size_t> size` in our class which gets initialized to 1 for every element. When merging two sets, the larger set becomes the parent of the smaller set and we sum up the two sizes:

```
private:
    ...
    std::vector<size_t> size;

public:
    union_find(size_t n) : parent(n), size(n, 1) { ... }

    ...

    void merge(size_t i, size_t j) {
        size_t pi = find(i);
        size_t pj = find(j);
        if (pi == pj) {            // If the elements are in the same set:
            return;                // do nothing
        }
        if (size[pi] > size[pj]) { // Swap representatives such that pj
            std::swap(pi, pj);     // represents the larger set
        }
        parent[pi] = pj;           // attach the smaller set to the larger one
        size[pj] += size[pi];      // update the size of the larger set
    }
```

Improvements: Union by rank

Another heuristic commonly used instead of union by size is the *union by rank* heuristic

Its basic idea is that we don't actually need to store the exact size of the sets, an approximation of the size (in this case: roughly the logarithm of the set's size) suffices to achieve the same speed as union by size.

For this, we introduce the notion of the *rank* of a set, which is given as follows:

- Singletons have rank 0
- If two sets with unequal rank are merged, the set with larger rank becomes the parent while the rank is left unchanged.
- If two sets of equal rank are merged, one of them becomes the parent of the other (the choice can be arbitrary), its rank is incremented.

One advantage of *union by rank* is the space usage: As the maximum rank increases roughly like $\log n$, for all realistic input sizes, the rank can be stored in a single byte (since `n < 2^255`).

A simple implementation of union by rank might look like this:

```
private:
    ...
    std::vector<unsigned char> rank;

public:
    union_find(size_t n) : parent(n), rank(n, 0) { ... }

    ...

    void merge(size_t i, size_t j) {
        size_t pi = find(i);
        size_t pj = find(j);
        if (pi == pj) {
            return;
        }
        if (rank[pi] < rank[pj]) {
            // link the smaller group to the larger one
            parent[pi] = pj;
        } else if (rank[pi] > rank[pj]) {
            // link the smaller group to the larger one
            parent[pj] = pi;
        } else {
            // equal rank: link arbitrarily and increase rank
            parent[pj] = pi;
            ++rank[pi];
        }
    }
```

Final Improvement: Union by rank with out-of-bounds storage

While in combination with path compression, union by rank nearly achieves constant time operations on union-find data structures, there is a final trick that allows us to get rid of the `rank` storage altogether by storing the rank in out-of-bounds entries of the `parent`array. It is based on the following observations:

- We actually only need to store the rank for *representatives*, not for other elements. For these representatives, we don't need to store a `parent`.
- So far, `parent[i]` is at most `size - 1`, i.e. larger values are unused.
- All ranks are at most $\log n$.

This brings us to the following approach:

- Instead of the condition `parent[i] == i`, we now identify representatives by `parent[i] >= size`
- We use these out-of-bounds values to store the ranks of the set, i.e. the set with representative `i`has rank `parent[i] - size`
- Thus we initialize the parent array with `parent[i] = size` instead of `parent[i] = i`, i.e. each set is its own representative with rank 0.

Since we only offset the rank values by `size`, we can simply replace the `rank` vector by the `parent` vector in the implementation of `merge` and only need to exchange the condition identifying representatives in `find`:

Finished implementation using union by rank and path compression:

```
using std::size_t;

class union_find {
private:
    std::vector<size_t> parent;

public:
    union_find(size_t n) : parent(n, n) {} // initialize with parent[i] = n

    size_t find(size_t i) const {
        if (parent[i] >= parent.size()) // If we already have a representative
            return i;                   // return it
        return find(parent[i]);         // otherwise return the parent's repr.
    }

    void merge(size_t i, size_t j) {
        size_t pi = find(i);
        size_t pj = find(j);
        if (pi == pj) {
            return;
        }
        if (parent[pi] < parent[pj]) {
            // link the smaller group to the larger one
            parent[pi] = pj;
        } else if (parent[pi] > parent[pj]) {
            // link the smaller group to the larger one
            parent[pj] = pi;
        } else {
            // equal rank: link arbitrarily and increase rank
            parent[pj] = pi;
            ++parent[pi];
        }
    }
};
```

Read Union-find data structure online: https://riptutorial.com/data-structures/topic/10684/union-find-data-structure

Credits

S. No	Chapters	Contributors
1	Getting started with data-structures	Community, Keshav Sharma, Vardan
2	Binary Heap	Vishwas
3	Binary Search Tree	Redet Getachew, venkatvb
4	Deque(Double ended queue)	Isha Agarwal
5	Graph traversals	BPD1
6	Linked List	Ashish Ahuja, Community, Ravindra HV
7	Matrices	Community
8	Queue	Community, Isha Agarwal, lambda
9	Segment Tree	Bakhtiar Hasan, Hesham Attia
10	Stack	Anup Kumar Gupta, Community, I. Afrin, Isha Agarwal, Jim Mischel, Vardan
11	Trie (Prefix Tree/Radix Tree)	Bakhtiar Hasan, RamenChef
12	Union-find data structure	Tobias Ribizel