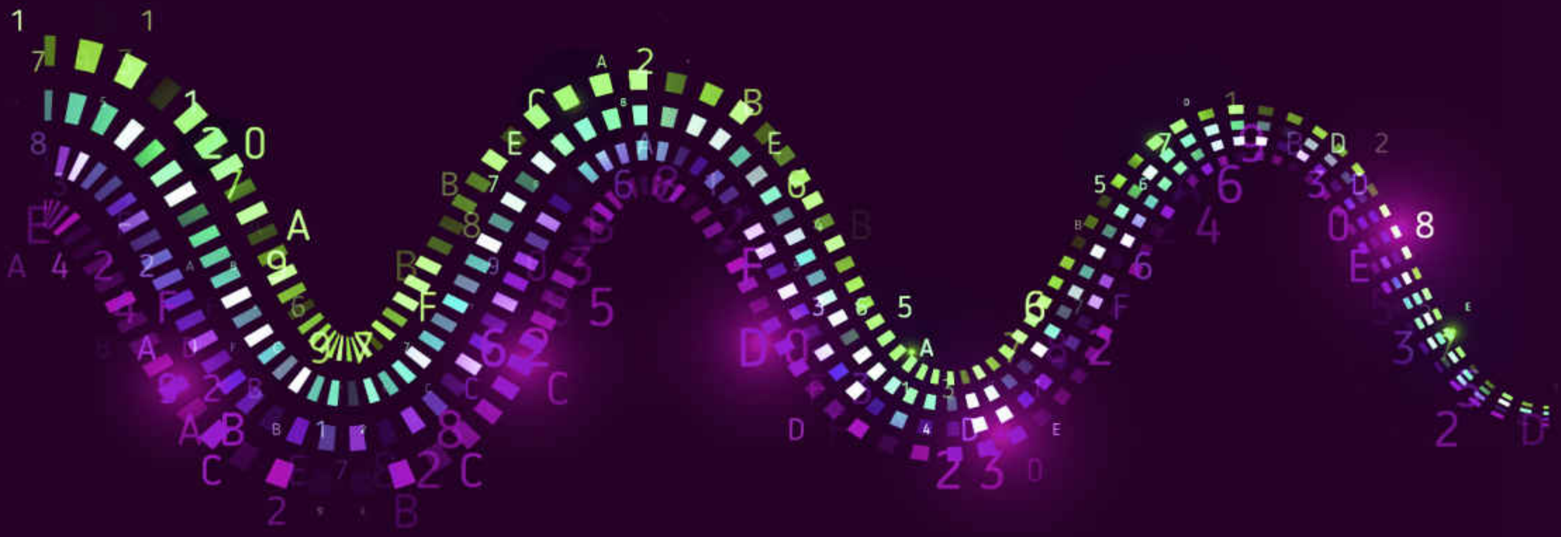
ALGORITHMS



ADVANCED DATA STRUCTURES FOR ALGORITHMS

ANDY VICKLER

Algorithms

Advanced Data Structures for Algorithms

Table of Contents

Introduction

Data structures and algorithms go together like a hand in a glove – algorithms need data structures to work. This guide will take you through some of the more advanced data structures that help you solve even more complex problems. Naturally, I assume that you already have knowledge of basic data structures and how they work in algorithms.

After you have studied this guide, you should find it easier to spot where your code can be improved in performance terms by using a more advanced data structure.

I don't expect you to learn everything in this book the first time you read it – that would be almost impossible. Rather, I would like you to use this as a guide to fall back on when you want to know if a certain data structure will fit your situation. I have provided lots of code examples, but because this is not specifically a computer programming book, I have used different programming languages to show you that you are not stuck with a specific language.

Here's what you will find in this guide:

- An overview of linked lists before moving on to doubly linked lists, XOR lists, self-organizing lists, and unrolled lists
- Some of the more advanced tree data structures, such as segment, trie, Fenwick, AVL, red-black, scapegoat, N-ary, and treap
- A discussion on disjoint sets
- The final section is dedicated to heaps and priority queues, including an overview of binary heaps before discussing binomial heaps, Fibonacci heaps, leftish heaps, K-ary heaps, and iterative heapsorts.

Please, please do not expect to grasp this the first time around. Take your time, go through each section in order and make sure you have a basic understanding before moving on to the next section.

Part 1: Advanced Lists

Linked Lists

Linked lists are similar to arrays in that they are linear data structures. However, where they differ is that elements in a linked list are not stored in contiguous locations. Instead, pointers are used to link them.

So, why use a linked list rather than an array?

Simply because, while an array can store similar types of linear data, they do have some limitations:

1. **The array has a fixed size.** This means we need to know the maximum number of elements it can take upfront. Also, regardless of the use, the allocated memory is typically equal to the maximum number.
2. **Inserting new elements is expensive.** Room must be made for new elements, which means existing ones must be moved. For example, let's say we have a list of sorted IDs stored in an array called **id[].**

   id[] = [1000, 1020, 1040, 1060]

   We want a new ID of 1010 inserted, but the sorted order must be maintained. All the elements following (but not including) 1000 need to be moved to do that. By the same token, deletion is also expensive because of the need to move everything after the deleted point.

Linked lists have a couple of advantages over arrays:

1. They have a dynamic size
2. Inserting and deleting elements is easy

However, they also have a couple of drawbacks too:

1. **Random access cannot be done.** Elements must be accessed sequentially, starting from the beginning. So, binary search is not efficient with the default implementation of a linked list.
2. **Extra memory space is required.** Every element in the list requires additional space for its pointer.

3. **They aren’t cache-friendly**. Because the elements in an array are all contiguous locations, each has a reference locality, which isn’t available in a linked list.

Linked lists are represented by a pointer that points to the first node, which is also called the head. In the case of an empty linked list, the head has a NULL value. Each node in a linked list has two or more parts:

1. Data
2. A pointer, or a reference to the following node

Nodes are represented in C language by data structures, while in C# or Java, a class represents a LinkedList, and Nodes are separate classes. The LinkedList class always has a reference that is of the Node class type.

That was an overview of linked lists, just a reminder, so the next topic makes sense.

Doubly Linked List

Doubly Linked Lists, or DLLs for short, have an additional pointer in them, usually known as a previous pointer. This joins the data and the next pointer in standard linked lists.

Below, the DLL node is represented in C:

```
/* Node of a doubly linked list */
class Node
{
    public:
    int data;
    Node* next; // Pointer to next node in DLL
    Node* prev; // Pointer to previous node in DLL
};
```

Doubly linked lists have some advantages over singly linked lists:

1. We can traverse a DLL forward and backward
2. Delete operations are more efficient when the node pointer we want to be deleted is given
3. New nodes can easily and quickly be inserted.

To delete nodes from singly linked lists, we need the pointer to the previous node, and sometimes we have to traverse the list to get this. In doubly linked lists, the previous pointer can be used to obtain the previous node.

DLLs also have a couple of disadvantages:

1. All DLL nodes need additional space for a previous pointer.
2. All operations require the extra previous pointer to be maintained. In insertion, for example, the previous pointers and the next pointers must be modified. In the following insertion functions, setting the previous pointer requires additional steps:

Insertions:

These can be done in four ways:

1. At the front
2. After a specified node
3. At the end
4. Before a specified node

Here are those insertions:

1. **Adding a new node at the front.**

New nodes are always added before the linked list's head, with the new node becoming the new head. For example, if we have a list of 123456789 and add a new node of 0 at the front, that list will become 0123456789. The function used to add the new node is called push(), and this must be given a pointer to the head pointer. This is because the push() function changes the head pointer to point to the new node.

The steps needed to add the node at the front are shown below:

```
/* Given a reference (pointer to pointer)
to the head of a list
and an int, inserts a new node on the
front of the list. */
void push(Node** head_ref, int new_data)
{
    /* 1. allocate node */
    Node* new_node = new Node();

    /* 2. put in the data */
    new_node->data = new_data;

    /* 3. Make next of new node as head
    and previous as NULL */
```

```
        new_node->next = (*head_ref);
        new_node->prev = NULL;

        /* 4. change prev of head node to new node */
        if ((*head_ref) != NULL)
            (*head_ref)->prev = new_node;

        /* 5. move the head to point to the new node */
        (*head_ref) = new_node;
    }
```

Four of those steps are identical to those used to insert a new node at the front of a singly linked list. The additional step changes the head's previous pointer.

2. Adding a new node after a given one.

We have a pointer to a node called prev_node, and the new node is to be inserted after this.

Below are the steps needed to do this:

```
/* Given a node as prev_node, insert
a new node after the given node */
void insertAfter(Node* prev_node, int new_data)
{
    /*1. check if the given prev_node is NULL */
    if (prev_node == NULL)
    {
       cout<<"the given previous node cannot be NULL";
        return;
    }
```

```
        /* 2. allocate new node */
        Node* new_node = new Node();

        /* 3. put in the data */
        new_node->data = new_data;

        /* 4. Make next of new node as next of prev_node */
        new_node->next = prev_node->next;

        /* 5. Make the next of prev_node as new_node */
        prev_node->next = new_node;

        /* 6. Make prev_node as previous of new_node */
        new_node->prev = prev_node;

        /* 7. Change previous of new_node's next node */
        if (new_node->next != NULL)
            new_node->next->prev = new_node;
    }
```

Five of these steps are identical to those used to insert a new node after a given one in the singly linked list. The additional steps change the new node's previous pointer and that of the new node's next node.

3. Adding nodes at the end.

The new node is added at the end of the given linked list. For example, our DLL is 0123456789, and we add an item, 30, at the end, the DLL will be 012345678930.

Because the head usually represents a linked list, the list must be traversed all the way through to the end, and the next-to-last-node is changed to the new node.

Again, here are the steps to do this:

```
/* Given a reference (pointer to pointer) to the head
of a DLL and an int, appends a new node at the end */
void append(Node** head_ref, int new_data)
{
    /* 1. allocate node */
    Node* new_node = new Node();

    Node* last = *head_ref; /* used in step 5*/

    /* 2. put in the data */
    new_node->data = new_data;

    /* 3. This new node is going to be the last node, so
       make next of it as NULL*/
    new_node->next = NULL;

    /* 4. If the Linked List is empty, then make the new
       node as head */
    if (*head_ref == NULL)
    {
       new_node->prev = NULL;
       *head_ref = new_node;
       return;
    }
    /* 5. Else traverse till the last node */
    while (last->next != NULL)
        last = last->next;

    /* 6. Change the next of last node */
    last->next = new_node;

    /* 7. Make last node as previous of new node */
    new_node->prev = last;

    return;
}
```

Six of these steps are identical to those to insert a new node after a specified node in a singly linked list, while the additional step changes the new node's previous pointer.

4. **Adding a node at the end**

Let's assume that the pointer to the given node is called next_node and the new node's data is added as new_data. The steps required are:

1. Determine whether next_node is NULL. If yes, the function must be returned from because new nodes cannot be added before NULLs.
2. Allocate the new node some memory
3. Set new_node->data = new_data
4. The new node's previous pointer should also be set as the next_node's previous node – new_node->prev = next_node->prev
5. The next_node's previous pointer should be set as new_node – new_node->prev = new_node
6. The new_node's pointer should be set at next_node – new_node->next = next_node

7. If new_node's previous node is not NULL, the previous node's pointer should be set as new_node – new_node->prev->next = new_node. If it is NULL, it becomes the head node so (*head_ref) = new_node.

This is how this approach is implemented:

Code block

Output:

```
Created DLL is:
Traversal in the forward Direction
9 1 5 7 6
Traversal in the reverse direction
6 7 5 1 9
```

The program below tests the functions:

```
// A complete working C++ program to
// demonstrate all insertion methods
#include <bits/stdc++.h>
using namespace std;

// A linked list node
class Node
{
    public:
    int data;
    Node* next;
    Node* prev;
};

/* Given a reference (pointer to pointer)
```

```
to the head of a list
and an int, inserts a new node on the
front of the list. */
void push(Node** head_ref, int new_data)
{
    /* 1. allocate node */
    Node* new_node = new Node();

    /* 2. put in the data */
    new_node->data = new_data;

    /* 3. Make next of new node as head
    and previous as NULL */
    new_node->next = (*head_ref);
    new_node->prev = NULL;

    /* 4. change prev of head node to new node */
    if ((*head_ref) != NULL)
        (*head_ref)->prev = new_node;

    /* 5. move the head to point to the new node */
    (*head_ref) = new_node;
}

/* Given a node as prev_node, insert
a new node after the given node */
void insertAfter(Node* prev_node, int new_data)
```

```
{
   /*1. check if the given prev_node is NULL */
   if (prev_node == NULL)
   {
      cout<<“the given previous node cannot be NULL”;
      return;
   }

   /* 2. allocate new node */
   Node* new_node = new Node();

   /* 3. put in the data */
   new_node->data = new_data;

   /* 4. Make next of new node as next of prev_node */
   new_node->next = prev_node->next;

   /* 5. Make the next of prev_node as new_node */
   prev_node->next = new_node;

   /* 6. Make prev_node as previous of new_node */
   new_node->prev = prev_node;

   /* 7. Change previous of new_node’s next node */
   if (new_node->next != NULL)
      new_node->next->prev = new_node;
}
```

```
/* Given a reference (pointer to pointer) to the head
of a DLL and an int, appends a new node at the end */
void append(Node** head_ref, int new_data)
{
    /* 1. allocate node */
    Node* new_node = new Node();

    Node* last = *head_ref; /* used in step 5*/

    /* 2. put in the data */
    new_node->data = new_data;

    /* 3. This new node is going to be the last node, so
        make next of it as NULL*/
    new_node->next = NULL;

    /* 4. If the Linked List is empty, then make the new
        node as head */
    if (*head_ref == NULL)
    {
        new_node->prev = NULL;
        *head_ref = new_node;
        return;
    }

    /* 5. Else traverse till the last node */
    while (last->next != NULL)
        last = last->next;

    /* 6. Change the next of last node */
    last->next = new_node;

    /* 7. Make last node as previous of new node */
    new_node->prev = last;

    return;
}

// This function prints contents of
// linked list starting from the given node
void printList(Node* node)
{
    Node* last;
    cout<<"\nTraversal in forward direction \n";
    while (node != NULL)
    {
        cout<<" "<<node->data<<" ";
        last = node;
        node = node->next;
    }

    cout<<"\nTraversal in reverse direction \n";
    while (last != NULL)
    {
        cout<<” “<<last->data<<” “;
        last = last->prev;
    }
}

/* Driver program to test above functions*/
int main()
{
    /* Start with the empty list */
    Node* head = NULL;

    // Insert 6. So linked list becomes 6->NULL
    append(&head, 6);

    // Insert 7 at the beginning. So
    // linked list becomes 7->6->NULL
    push(&head, 7);

    // Insert 1 at the beginning. So
    // linked list becomes 1->7->6->NULL
    push(&head, 1);

    // Insert 4 at the end. So linked
    // list becomes 1->7->6->4->NULL
    append(&head, 4);
```

```
        // Insert 8, after 7. So linked
        // list becomes 1->7->8->6->4->NULL
        insertAfter(head->next, 8);

        cout << “Created DLL is: “;
        printList(head);

        return 0;
    }
```

The output should be:

```
    Created DLL is:
    Traversal in the forward Direction
    1 7 5 8 6 4
    Traversal in the reverse direction
    4 6 8 5 7 1
```

XOR Linked Lists

Standard doubly linked lists need space where two address fields store the previous and next node addresses. The previous node address is retained, being carried over so the previous pointer can compute it.

The XOR linked list is a memory-efficient doubly linked list, and these are created using a single space for each node's address field using a bitwise XOR operation. In these lists, the nodes each store the previous and next nodes' XOR of addresses.

There are two ways that XOR representations behave differently from ordinary representations.

1. **Ordinary Representation**

   Node A:

   prev = NULL, next = add(B) // previous is NULL while next is address of B

   Node B:

   prev = add(A), next = add(C) // previous is address of A while next is address of C

   Node C:

   prev = add(B), next = add(D) // previous is address of B while next is address of D

   Node D:

   prev = add(C), next = NULL // previous is address of C while next is NULL

2. **XOR List Representation**

We'll call the XOR address variable npx. We can traverse an XOR linked list forward and backward, but, as we do, we must remember each previously accessed node's address so we can calculate the address for the next node.

For example, at node C, we need node B's address. An XOR of add(B) and an npx of C provides add(D).

Here's an illustration of that:

```
Node A:

npx = 0 XOR add(B) // bitwise XOR of zero and address of B


Node B:

npx = add(A) XOR add(C) // bitwise XOR of address of A and address of C


Node C:

npx = add(B) XOR add(D) // bitwise XOR of address of B and address of D


Node D:

npx = add(C) XOR 0 // bitwise XOR of address of C and 0

npx(C) XOR add(B)

=> (add(B) XOR add(D)) XOR add(B) // npx(C) = add(B) XOR add(D)

=> add(B) XOR add(D) XOR add(B) // a^b = b^a and (a^b)^c = a^(b^c)

=> add(D) XOR 0  // a^a = 0

=> add(D)     // a^0 = a
```

In the same way, the list can be traversed backward.

Here's an example of how to implement this:

```
// C++ Implementation of Memory
// efficient Doubly Linked List

// Importing libraries
#include <bits/stdc++.h>
#include <cinttypes>

using namespace std;

// Class 1
// Helepr class(Node structure)
class Node {
    public : int data;
    // Xor of next node and previous node
    Node* xnode;
};

// Method 1
// The Xored value of the node addresses is returned
Node* Xor(Node* x, Node* y)
{
    return reinterpret_cast<Node*>(
        reinterpret_cast<uintptr_t>(x)
        ^ reinterpret_cast<uintptr_t>(y));
}

// Method 2
// Insert a node at the beginning of the Xored LinkedList
and
// mark it as head
void insert(Node** head_ref, int data)
{
    // Allocate memory for new node
    Node* new_node = new Node();
    new_node -> data = data;

    // Since the new node has been inserted at the
    // start , the xnode of the new node will always be
    // Xor of current head and NULL
    new_node -> xnode = *head_ref;

    // If linkedlist is not empty, the xnode of the
    // present head node will be the Xor of the new node
    // and the node next to the current head */
    if (*head_ref != NULL) {
        // *(head_ref)->xnode is Xor of (NULL and next).
        // If we Xor Null with next we get next
        (*head_ref)
                -> xnode = Xor(new_node, (*head_ref) ->
xnode);
    }

    // Change head
    *head_ref = new_node;
}
```

```
// Method 3
// This prints the contents of the doubly linked
// list in a forward direction
void printList(Node* head)
{
    Node* curr = head;
    Node* prev = NULL;
    Node* next;

    cout << “The nodes of the Linked List are: \n”;

    // Until the condition holds true
    while (curr != NULL) {
       // print current node
       cout << curr -> data << ” “;

       // get the address of the next node: curr->xnode is
       // next^prev, so curr->xnode^prev will be
       // next^prev^prev which is next
       next = Xor(prev, curr -> xnode);

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

```
// Method 4
// main driver method
int main()
{
    Node* head = NULL;
    insert(&head, 10);
    insert(&head, 100);
    insert(&head, 1000);
    insert(&head, 10000);

    // Printing the created list
    printList(head);

    return (0);
}
```

Output:

```
10000 1000 100 10
```

So, we now know how to create a doubly linked list with each node having its address field in a single space. Now we can discuss how to implement these lists. We'll be talking about two functions:

3. One that inserts a new node at the start
4. One that traverses the list forwards.

In the next code example, we insert a new node at the start using the insert() function. The head pointer needs to be changed, which is why we use a double-pointer. Here are some things to remember:

- The XOR of the next and previous nodes are stored with each node and called npx.

- npx is the only address member with each node.
- When a new node is inserted at the beginning, the new node’s npx is always the XOR of NULL and the current head.
- The current head’s npx must be changed to the XOR or the new node and the node beside the current head.
- We use printList for forward traversal, printing the data values in each node. The pointer to the next node must be obtained at each point.
- The next node address can be obtained by tracking the current and previous nodes. An XOR of curr->npx and prev gives us the next node’s address.

Here’s a code example:

```
/* C++ Implementation of Memory
efficient Doubly Linked List */
#include <bits/stdc++.h>
#include <cinttypes>
using namespace std;

// Node structure of a memory
// efficient doubly linked list
class Node
{
    public:
    int data;
     Node* npx; /* The XOR of the next and previous
node */
};

/* returns the XORed value of the node addresses */
Node* XOR (Node *a, Node *b)
```

```
{
    return reinterpret_cast<Node *>(
      reinterpret_cast<uintptr_t>(a) ^
      reinterpret_cast<uintptr_t>(b));
}

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

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

    /* If the linked list is not empty, the npx of the
    current head node will be the XOR of the new node
    and the node next to the 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
         (*head_ref)->npx = XOR(new_node, (*head_ref)-
>npx);
    }

    // Change head
    *head_ref = new_node;
}

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

    cout << “Following are the nodes of Linked List: \n”;

    while (curr != NULL)
    {
        // print current node
        cout<<curr->data<<” “;

        // get the address of the 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 code
    int main ()
    {
        /* Create the following Doubly Linked List
        head—>40<—>30<—>20<—>10 */
        Node *head = NULL;
        insert(&head, 10);
        insert(&head, 20);
        insert(&head, 30);
        insert(&head, 40);

        // print the created list
        printList (head);

        return (0);
    }
```

Output:

```
The Following are the nodes of Linked List:
40 30 20 10
```

Note – C and C++ standards do not define the XOR of pointers so this may not work on every platform.

Self-Organizing Lists

Self-organizing lists are that reorder their elements based on a heuristic that helps improve access times. The aim is to improve linear search efficiency by moving those items we access more frequently towards the head or front of the list. In terms of element access, the best-case scenario for a self-organizing list is near-constant time.

There are two search possibilities – online, where we have no idea of the search sequence, and offline, where we have knowledge of the entire search sequence upfront. In the latter, the nodes may be placed per the decreasing search frequencies, where the element searched for the most frequently is placed first, and the least searched element goes last. In terms of real-world applications, knowing the search sequence upfront is not always easy to do. Self-organizing lists reorder themselves based on each search, so the aim is to use a locality of reference. In most databases, 20% of items are located, with 80% of the searches.

Below, we take a quick look at three strategies a self-organizing list uses.

1. **Move-To-Front Method**

This method moves the most recent item searched for to the front of the list, making it easy to implement. However, because it focuses on the recent search, even infrequently searched-for items will move to the head of the list, which is a disadvantage in terms of access time.

Here are some examples:

```
Input :   list : 1, 2, 3, 4, 5, 6
          searched: 4
Output :  list : 4, 1, 2, 3, 5, 6


Input :   list : 4, 1, 2, 3, 5, 6
          searched : 2
```

Output : list : 2, 4, 1, 3, 5, 6

```
// CPP Program to implement self-organizing list
// using move to front method
#include <iostream>
using namespace std;

// structure for self organizing list
struct self_list {
    int value;
    struct self_list* next;
};

// head and rear pointing to start and end of list resp.
self_list *head = NULL, *rear = NULL;

// function to insert an element
void insert_self_list(int number)
{
    // creating a node
    self_list* temp = (self_list*)malloc(sizeof(self_list));

    // assigning value to the created node;
    temp->value = number;
    temp->next = NULL;

    // first element of list
    if (head == NULL)
        head = rear = temp;

    // rest elements of list
    else {
        rear->next = temp;
        rear = temp;
    }
}

// function to search the key in list
// and re-arrange self-organizing list
bool search_self_list(int key)
{
    // pointer to current node
    self_list* current = head;

    // pointer to previous node
    self_list* prev = NULL;

    // searching for the key
    while (current != NULL) {

        // if key found
        if (current->value == key) {

            // if key is not the first element
            if (prev != NULL) {

                /* re-arranging the elements */
                prev->next = current->next;
                current->next = head;
                head = current;
            }
            return true;
        }
        prev = current;
        current = current->next;
    }

    // key not found
    return false;
}

// function to display the list
void display()
{
    if (head == NULL) {
        cout << “List is empty” << endl;
        return;
    }

    // temporary pointer pointing to head
    self_list* temp = head;
    cout << “List: “;

    // sequentially displaying nodes
    while (temp != NULL) {
        cout << temp->value;
        if (temp->next != NULL)
            cout << ” —> “;

        // incrementing node pointer.
        temp = temp->next;
    }
    cout << endl  << endl;
}

// Driver Code
int main()
{
    /* inserting five values */
    insert_self_list(1);
    insert_self_list(2);
    insert_self_list(3);
    insert_self_list(4);
    insert_self_list(5);

    // Display the list
    display();
```

```
        // search 4 and if found then re-arrange
        if (search_self_list(4))
            cout << “Searched: 4” << endl;
        else
            cout << “Not Found: 4” << endl;

        // Display the list
        display();

        // search 2 and if found then re-arrange
        if (search_self_list(2))
            cout << “Searched: 2” << endl;
        else
            cout << “Not Found: 2” << endl;
        display();

        return 0;
    }
```

Output:

```
List: 1 —> 2 —> 3 —> 4 —> 5

Searched: 4
List: 4 —> 1 —> 2 —> 3 —> 5

Searched: 2
List: 2 —> 4 —> 1 —> 3 —> 5
```

2. **The Count Method**

The count method is where a count is kept for the number of times a node is searched for, which means it records the search frequency. Each node has additional storage space associated with it, and this increments each time the node is searched. The nodes are then arranged in an order whereby the most searched goes to the front of the list.

Here are some examples:

```
Input : list : 1, 2, 3, 4, 5
        searched : 4
Output : list : 4, 1, 2, 3, 5


Input : list : 4, 1, 2, 3, 5
        searched : 5
        searched : 5
        searched : 2
```

Output :

```
list : 5, 2, 4, 1, 3
```

Let's explain this:

- 5 is the most searched-for – twice – which puts it at the head of the list.
- 2 is searched for once, as is 1
- Because 2 is searched for more recently than 1, it is kept ahead of 4
- Because the remainder are not searched for, they retain the order they were inserted into the list.

```
// CPP Program to implement self-organizing list
// using thecount method
#include <iostream>
using namespace std;
```

```
// structure for self organizing list
struct self_list {
    int value;
    int count;
    struct self_list* next;
};

// head and rear pointing to start and end of list resp.
self_list *head = NULL, *rear = NULL;

// function to insert an element
void insert_self_list(int number)
{
    // creating a node
    self_list* temp = (self_list*)malloc(sizeof(self_list));

    // assigning value to the created node;
    temp->value = number;
    temp->count = 0;
    temp->next = NULL;

    // first element of list
    if (head == NULL)
        head = rear = temp;

    // rest elements of list
    else {
        rear->next = temp;
        rear = temp;
    }
}

// function to search the key in list
// and re-arrange self-organizing list
bool search_self_list(int key)
{
    // pointer to current node
    self_list* current = head;

    // pointer to previous node
    self_list* prev = NULL;

    // searching for the key
    while (current != NULL) {

        // if key is found
        if (current->value == key) {

            // increment the count of node
            current->count = current->count + 1;

            // if it is not the first element
            if (current != head) {
```

```
            self_list* temp = head;
            self_list* temp_prev = NULL;

            // finding the place to arrange the searched node
            while (current->count < temp->count) {
                temp_prev = temp;
                temp = temp->next;
            }

            // if the place is other than its own place
            if (current != temp) {
                prev->next = current->next;
                current->next = temp;

                // if it is to be placed at beginning
                if (temp == head)
                    head = current;
                else
                    temp_prev->next = current;
            }
        }
        return true;
    }
    prev = current;
    current = current->next;
}
return false;
```

```
}

// function to display the list
void display()
{
    if (head == NULL) {
        cout << “List is empty” << endl;
        return;
    }

    // temporary pointer pointing to head
    self_list* temp = head;
    cout << “List: “;

    // sequentially displaying nodes
    while (temp != NULL) {
        cout << temp->value << “(” << temp->count << “)”;
        if (temp->next != NULL)
            cout << ” —> “;

        // incrementing node pointer.
        temp = temp->next;
    }
    cout << endl
         << endl;
}
```

```
// Driver Code
int main()
{
    /* inserting five values */
    insert_self_list(1);
    insert_self_list(2);
    insert_self_list(3);
    insert_self_list(4);
    insert_self_list(5);

    // Display the list
    display();

    search_self_list(4);
    search_self_list(2);
    display();

    search_self_list(4);
    search_self_list(4);
    search_self_list(5);
    display();

    search_self_list(5);
    search_self_list(2);
    search_self_list(2);
    search_self_list(2);
    display();
```

```
        return 0;
    }
```

Output :

```
List: 1(0) —> 2(0) —> 3(0) —> 4(0) —> 5(0)
List: 2(1) —> 4(1) —> 1(0) —> 3(0) —> 5(0)
List: 4(3) —> 5(1) —> 2(1) —> 1(0) —> 3(0)
List: 2(4) —> 4(3) —> 5(2) —> 1(0) —> 3(0)
```

3. **The Transpose Method**

The transpose method swaps the accessed node with its predecessor. This means when a node is accessed, it gets swapped with the one in front. The only exception to this is when the accessed node is the head. Put simply, an accessed node's priority is slowly increased until it reaches the head position. The difference between this and the other methods is that it requires many accesses to get a node to the head.

Here are some examples:

Input :

```
list: 1, 2, 3, 4, 5, 6
        searched: 4
```

Output :

```
list: 1, 2, 4, 3, 5, 6
```

Input :

```
list: 1, 2, 4, 3, 5, 6
        searched: 5
```

Output :

```
list: 1, 2, 4, 5, 3, 6
```

Let's explain this:

- In the first case, node 4 is swapped with node 3, its predecessor
- In the second case, node 5 is swapped with its predecessor, also 3

```
// CPP Program to implement self-organizing list
// using the move to front method
#include <iostream>
using namespace std;

// structure for self organizing list
struct self_list {
    int value;
    struct self_list* next;
};

// head and rear pointing to start and end of list resp.
self_list *head = NULL, *rear = NULL;

// function to insert an element
void insert_self_list(int number)
{
    // creating a node
    self_list* temp = (self_list*)malloc(sizeof(self_list));

    // assigning value to the created node;
    temp->value = number;
    temp->next = NULL;
    // first element of list
    if (head == NULL)
        head = rear = temp;

    // rest elements of list
    else {
        rear->next = temp;
        rear = temp;
    }
}

// function to search the key in list
// and re-arrange self-organizing list
bool search_self_list(int key)
{
    // pointer to current node
    self_list* current = head;

    // pointer to previous node
    self_list* prev = NULL;

    // pointer to previous of previous
    self_list* prev_prev = NULL;

    // searching for the key
    while (current != NULL) {
        // if key found
        if (current->value == key) {

            // if key is neither the first element
            // and nor the second element
            if (prev_prev != NULL) {

                /* re-arranging the elements */
                prev_prev->next = current;
                prev->next = current->next;
                current->next = prev;
            }

            // if key is second element
            else if (prev != NULL) {

                /* re-arranging the elements */
                prev->next = current->next;
                current->next = prev;
                head = current;
            }
            return true;
        }
        prev_prev = prev;
        prev = current;
        current = current->next;
    }
    // key not found
    return false;
}

// function to display the list
void display()
{
    if (head == NULL) {
        cout << “List is empty” << endl;
        return;
    }

    // temporary pointer pointing to head
    self_list* temp = head;
    cout << “List: “;

    // sequentially displaying nodes
    while (temp != NULL) {
        cout << temp->value;
        if (temp->next != NULL)
            cout << ” —> “;

        // incrementing node pointer.
        temp = temp->next;
    }
    cout << endl
```

```
        << endl;
}

// Driver Code
int main()
{
    /* inserting five values */
    insert_self_list(1);
    insert_self_list(2);
    insert_self_list(3);
    insert_self_list(4);
    insert_self_list(5);
    insert_self_list(6);

    // Display the list
    display();

    // search 4 and if found then re-arrange
    if (search_self_list(4))
        cout << “Searched: 4” << endl;
    else
        cout << “Not Found: 4” << endl;

    // Display the list
    display();

    // search 2 and if found then re-arrange
```

```
        if (search_self_list(5))
            cout << “Searched: 5” << endl;
        else
            cout << “Not Found: 5” << endl;
        display();

        return 0;
    }
```

Output:

```
    List: 1 —> 2 —> 3 —> 4 —> 5 —> 6

    Searched: 4
    List: 1 —> 2 —> 4 —> 3 —> 5 —> 6

    Searched: 5
    List: 1 —> 2 —> 4 —> 5 —> 3 —> 6
```

Unrolled Linked List

Unrolled linked list data structures are a linked list variant. Rather than a single element being stored at a node, these will store entire arrays.

With this type of list, you get the benefits of arrays in terms of the minor memory overhead, with the benefits of linked lists, in terms of fast deletion and insertion. Those benefits combine to give much better performance.

The unrolled linked list spreads the overheads (pointers) by storing several elements at each node. So, if an array containing 4 elements is stored at each node, the overhead is spread across those elements.

Unrolled linked lists also perform much faster when you consider the modern CPU's capacity for cache management. So, while their overhead is quite high per node compared to a standard linked list, this is a minor drawback compared to the benefits it offers in modern computing.

Properties

Unrolled link lists are basically linked lists where an array of values is stored in a node rather than a single value. The array can have anything a standard array has, like abstract data types such as primitive types. Each node can only hold a certain number of values, and the average implementation will ensure each node has an average capacity of 3/43/3. This is achieved by the node moving values from one array to another when one gets full.

An unrolled link list has a slightly higher overhead per node because each must also store the maximum number of values per array. However, on a per value basis, it actually works out lower than standard linked lists. As each array's maximum size increases, the average space required for each value reduces and, when the value is a bit or another very small type, you get even more space advantages.

In short, an unrolled linked list is a combination of a standard linked list and an array. You get the ultra-fast indexing and storage locality advantages of the array, both of which arise from the static array's underlying properties. On top of that, you retain the node insertion and deletion advantages from the linked list.

Insertion and Deletion

The algorithm used to insert or delete elements in unrolled linked lists will depend on the implementation. The low-water mark is typically around 50%, which means when an insertion is carried out in a node that doesn't have space for the element, a new node gets created, and 50% of the elements from the original array are inserted. Conversely, if an element is removed, resulting in the number of values reducing to under 50% in the node, the elements from the array next door are moved in to push it back up to 50%. Should that result in the neighboring array dropping below 50%, both nodes are merged.

Generally, this means the average node utilization is 70-75%, and, with reasonable node sizes, the overhead is small compared to a standard linked list. In standard linked lists, 2 to 3 pieces of overhead are required per node, but the unrolled linked lists amortize this across the elements, resulting in 1/size1/size, nearer to an array's overhead.

The low-water mark can be altered to alter your list's performance. Increase it, and each node's average utilization increases, but there is a cost to this – splitting and merging would need to be done more frequently.

Algorithm Pseudocode

The insertion and deletion algorithm is as follows. The Node's all contain an array called data, several elements in the array numElements, and next, which is a pointer to the next node.

```
Insert(newElement)
    Find node in linked list e
    If e.numElements < e.data.size
```

```
            e.data.push(newElement)
            e.numElements ++
        else
            create new Node e1
            move the final half of e.data into e1.data
            e.numElements = e.numElements / 2
            e1.data.push(newElement)
            e1.numElements = e1.data.size / 2 + 1

    Delete(element)
        Find element in node e
        e.data.remove(element)
        e.numElements —
        while e.numElements < e.data.size / 2
            put element from e.next.data in e.data
            e.next.numElements —
            e.numElements ++
        if e.next.numElements < e.next.data.size / 2
            merge nodes e and e.next
            delete node e.next
```

A programmer can take quite a few liberties with these functions. For example, in the Insert function, the programmer can decide which node they want to insert into. It could be the first or last, or it could be determined by a specific grouping or sorting. Typically, this will depend entirely on the data the programmer is working with.

Time and Space Complexity

It isn't easy to analyze an unrolled linked list because there are many ways to implement them and plenty of data-dependent variations. However, their amortization across the array elements ensures their time and space analysis is good.

Time

When you want to insert an element, the first step is to locate the node you want the element to be inserted in, which is O(n)O(n). If the node isn't full, that's all there is to it. However, a new node needs to be created if the node is full and the values moved over. This process isn't dependent on the linked list's total values, so it is in constant time.

Deleting an element works in the same way but in reverse. Constant time operations can be taken advantage of because linked lists can quickly update their pointers between each other, independent of the list size.

Another quality of unrolled linked lists considered important is indexing. However, it depends on caching, which we'll cover in a moment.

Operation	**Complexity**
insertion	O(n)O(n)
deletion	O(n)O(n)
indexing	O(n)O(n)
search	O(n)O(n)

The most important thing to remember in unrolled linked lists is that practical application is where the true benefits lie, not in asymptotic analysis.

Space

How much space an unrolled linked list requires will wary from (v/m)n(v/m)n to 2(v/m)n2(v/m)n. Here, vv indicates each node's overhead, mm indicates each node's maximum number of elements, and nn indicates the number of nodes. While this is much the same as the asymptotic space other linear data structures use, it is a much better space because the overheads are spread over many elements.

	Complexity
space	O(n)O(n)

Caching

The unrolled link list's real benefits come in its caching, something the list uses for indexing.

When programs access data from memory, the entire page is pulled. If the right value isn't found, it is called a cache miss. This means an unrolled linked list can be indexed m/n + 1m/n+1 cache misses at the most, faster than standard linked lists by up to a factor of mm.

This can be analyzed further by considering the cache line size, called BB. Often, mm, which is each node's array size, is equal to or is a multiple of the cache line size needed for optimal fetching. We traverse each node in (n/B)(n/B) cache misses, and the list can be traversed in (m/n + 1)(n/B)(m/n+1)(n/B). This is as close as possible to the optimal cache miss value - (m/B)(m/B).

In Part 2, we'll look at some of the advanced tree structures.

Part 2: Advanced Trees

Segment Trees

We use segment trees when we need to do several rang queries on arrays while modifying the array elements at the same time. For example, we may want to find the sum of the array elements from indices L to R, or we may need to find the minimum of all the array elements from L to R – the latter is better known as the Range Minimum Query Problem. The segment tree is an incredibly versatile data structure and can solve these and more problems.

Basically, a segment tree is a binary tree that stores segments or intervals, and each tree node represents one interval. For example, we have an array A of size N and a segment tree T:

- The entire array A[o:N-1] is represented by the root of T
- Each leaf in T represents one element A[i] in a way that 0≤i<N
- T's internal nodes represent the union of the elementary intervals A[i:j] where 0≤i<j<N.

The segment tree's root represents A[0:N-1] which is the entire array. This is then broken into segments or two half intervals. The A[0:(N−1)/2] and A[(N−1)/2+1:(N−1)] are represented by the root's two children.

In each step, the segment gets divided into two, and the two children represent the halves. The segment tree's height is log2N, N leaves represent the N elements in the arrays, and N-1 is the number of the internal nodes. Therefore, the nodes total 2xN-1 in number.

You cannot change a segment tree's structure once you have built it, but you can update the node values. There are two operations a segment tree can do:

1. **Update -** updates the array A element and reflects the change in the tree
2. **Query** – used to query segments or intervals and return the answer to the specific problem, i.e., summation, minimum or maximum in the specified segment.

Implementing a Segment Tree:

Because this is a binary tree, it can be represented with a linear array. Before you build a segment tree, you need to work out what you want to store in its node. For example, if you want to find the sum of an array's elements from indices L to R, each node will store the sum of its own children nodes – the exception to this is the leaf nodes.

We can use recursion to build a segment tree, which is the bottom-up approach. Begin with the leaves and move up to the root (always at the top). Update the corresponding node changes on the path between the leaves and root. Each leaf represents one element, and an internal parental node is formed in each step by using the data from the two children nodes. These internal nodes each represent unions between the children's intervals. The merging may differ depending on the question, so recursion always ends at the root node, representing the entire array.

Updating requires searching the leaf with the element you want to be updated. We do this in one of two ways – going to the left or right child, depending on which interval has the relevant element. The leaf is updated when found, with the corresponding changes made in the leaf to the root path using the bottom-up approach.

A range from L to R needs to be selected to query on a tree – this is usually provided in the question. Begin at the root and recurse on the tree, checking if the interval that the node represents is completely in that range. If it is, the node's value is returned.

You can do queries and updates in any order you want.

How to Use a Segment Tree

First, you need to determine what is being stored in the tree node. For the purposes of this, we want the summation from the l to r interval. In each node, we need to sum the elements in the interval the node represents.

Next, the tree must be built. Below, you can see the implementation, which shows you the process needed to build the tree:

```
void build(int node, int start, int end)
{
    if(start == end)
    {
        // The leaf node has a single element
        tree[node] = A[start];
    }
    else
    {
        int mid = (start + end) / 2;
        // Recurse on the left child
        build(2*node, start, mid);
        // Recurse on the right child
        build(2*node+1, mid+1, end);
            // The internal node has the sum of both of its
children
        tree[node] = tree[2*node] + tree[2*node+1];
    }
}
```

As you can see in this code, we start at the root and recurse until we reach a leaf node. This is done on both the left and right child. Go back to the root from the leaves, updating all nodes along the path. The current node to be processed is indicated by .node. Because segment trees are binary, the left node is represented by 2xnode and the right by 2xnode+1. The interval the node represents is, in turn, represented by start and end, and the build complexity is O(N).

When you want to update an element, you need to look at the interval where the element is and recurse on the left child or the right one.

```
void update(int node, int start, int end, int idx, int val)
{
    if(start == end)
    {
        // Leaf node
        A[idx] += val;
        tree[node] += val;
    }
    else
   {
        int mid = (start + end) / 2;
        if(start <= idx and idx <= mid)
        {
            // If idx is in the left child, recurse on the left child
            update(2*node, start, mid, idx, val);
        }
        else
        {
            // if idx is in the right child, recurse on the right child
            update(2*node+1, mid+1, end, idx, val);
        }
        // The internal node will have the sum of both of its children
        tree[node] = tree[2*node] + tree[2*node+1];
        }
    }
```

The update complexity is O(logN).

To run a query on a specific range, 3 conditions must be checked:

1. Range that a node represents is entirely inside the specified range
2. Range that a node represents is entirely outside the specified range
3. Range that a node represents is partly in and partly outside the specified range.

With condition 1, 0 is returned. With condition 2, the value of the node returned is the sum of the elements in the represented node. With condition 3, the sum of the left and right children is returned.

The query complexity is O(logN).

```
int query(int node, int start, int end, int l, int r)
{
    if(r < start or end < l)
    {
        // range that a node represents is entirely outside the
specified range
        return 0;
    }
    if(l <= start and end <= r)
    {
        // range that a node represents is entirely inside the
specified range
        return tree[node];
    }
    // range that a node represents is partly in and partly outside the specified range.
    int mid = (start + end) / 2;
    int p1 = query(2*node, start, mid, l, r);
    int p2 = query(2*node+1, mid+1, end, l, r);
    return (p1 + p2);
}
```

Trie Data Structures

Trie comes from the word “retrieval,” and a trie is a sorted tree data structure used to store sets of strings. Its number of pointers is equal to how many alphabet characters are in each node. You can search for words in a given dictionary just by using the word’s prefix. For example, assuming that we form all the strings using the alphabet letters a to z, each node on the trie may have no more than 26 points.

A trie is also called a prefix or digital tree. Where a node is positioned in a trie determines the key the node is connected to.

Properties for a string set:

1. The trie’s root node will always represent the null node
2. Each of the node’s children is alphabetically sorted
3. Each node can have up to 26 children between a and z
4. Each node can store a single alphabet letter, except for the root.

Basic Operations

A trie has three basic operations:

1. Inserting a node
2. Search a node
3. Deleting a node

Inserting a Node

Before we look at how a node is inserted into a trie, there are a few points you need to understand:

1. Each letter of the word, or input key, is inserted individually in the Trie_node. The children point to the next Trie node level.
2. The key array of characters is the index of children.
3. If there is a reference to the present letter in the present node, the present node should be set to the referenced node. Otherwise, a new node should be created, the letter

set as equal to the present one, and the present node can be started with the new one.
4. The character length determines the trie's depth.

Implementing the insertion of a new node:

```
public class Data_Trie {
    private Node_Trie root;
    public Data_Trie(){
        this.root = new Node_Trie();
    }
    public void insert(String word){
        Node_Trie current = root;
        int length = word.length();
        for (int x = 0; x < length; x++){
            char L = word.charAt(x);
            Node_Trie node = current.getNode().get(L);
            if (node == null){
                node = new Node_Trie ();
                current.getNode().put(L, node);
            }
            current = node;
        }
        current.setWord(true);
    }
}
```

Searching a Node

To search a node in a trie is similar to inserting a node, although the operation searches a key. You can see how this is implemented below:

```
class Search_Trie {
    private Node_Trie Prefix_Search(String W) {
        Node_Trie node = R;
        for (int x = 0; x < W.length(); x++) {
            char curLetter = W.charAt(x);
            if (node.containsKey(curLetter))
             {
                node = node.get(curLetter);
              }
            else {
                return null;
            }
        }
        return node;
    }
    public boolean search(String W) {
        Node_Trie node = Prefix_Search(W);
        return node != null && node.isEnd();
    }
}
```

Deleting a Node

Again, before we look at implementing a deletion, there are a couple of things you need to understand:

1. If the delete operation cannot find the specified key, it will stop and exit
2. If the delete operation does find the key, it is deleted.

```
public void Node_delete(String W)
{
        Node_delete(R, W, 0);
    }

    private boolean Node_delete(Node_Trie current, String W, i
    nt Node_index) {
        if (Node_index == W.length()) {
            if (!current.isEndOfWord()) {
                return false;
            }
            current.setEndOfWord(false);
            return current.getChildren().isEmpty();
        }
        char A = W.charAt(Node_index);
        Node_Trie node = current.getChildren().get(A);
        if (node == null) {
            return false;
        }
        boolean Current_Node_Delete = Node_delete(node, W,
    Node_index + 1) && !node.isEndOfWord();

        if (Current_Node_Delete) {
            current.getChildren().remove(A);
            return current.getChildren().isEmpty();
        }
        return false;
    }
```

Applications of Trie

Tries have a few applications:

A Spell Checker

There are three steps to the spellchecking process:

- Step one – look for the required word in the dictionary
- Step two – generate a few suggestions
- Step three – sort those suggestions, ensuring the word you want is at the top

Trie stores the word in a dictionary, and the spell checker can be efficiently applied to find words in a data structure. Not only will you see the word easily in the dictionary, but you will also find it much simpler to build algorithms that include collections of relevant suggestions or words.

Auto-Complete

This function tends to be used widely on mobile apps, the internet, and text editors. It gives us an easy way to find alternative words to complete a word for these reasons:

- It gives an entries filter in alphabetical order by the node's key
- Pointers can be traced to get the node representing the user-entered string
- When you begin typing, auto-complete will attempt to complete what you write.

Browser History

Trie is also used to complete URLs in browsers. Your browser generally keeps a record of the websites you have already visited, allowing you to find the one you want easily.

Advantages

- It is much easier to insert, and strings are faster to search than standard binary trees and hash tables.
- Using the node's key, it can give you an entry filter in alphabetical order.

Disadvantages

- More memory is needed for string storage
- It isn't as fast as a hash table

Fenwick Tree

Fenwick trees, otherwise known as binary indexed trees (BIT), allow us to represent arrays of numbers in arrays and efficiently calculate prefix sums. Take an array of [2, 3, -1, 0,6] for example. The prefix sum is the sum of the first three elements [2, 3, -1] which is calculated as 2 + 3 + -1 = 4. Efficient prefix sum calculation is useful in several situations so let's begin with a simple problem.

Let's say we have an array a[], and we want to perform two different types of operation on it:

1. **Point Update Operation** – we want to modify a value that is stored at index i
2. **Range Sum Query** – we want the sum of a prefix which is of length k

Here you can see a simple implementation of this:

```
int a[] = {2, 1, 4, 6, -1, 5, -32, 0, 1};

void update(int i, int v)   //assigns value v to a[i]

{

    a[i] = v;

}

int prefixsum(int k)    //calculate the sum of all a[i] such that 0 <= i < k

{

  int sum = 0;

  for(int i = 0; i < k; i++)

       sum += a[i];

  return sum;

}
```

This is a great solution but with a downfall – the time needed for the prefix sum calculation is in proportion to the array length so, when we perform intermingled operations with such big numbers, it tends to time out.

Perhaps the most efficient way is with a segment tree as these can do both operations is O(logN) time.

However, we can also do both operations in O(logN) time with the Fenwick tree, but there is little sense in learning a new data structure when you already have the perfect one for it. BIT trees don't take so much space and are easier to program, given that they don't need any more than 10 lines of code.

Before we dig deeper into the BIT tree, let's look at a quick bit manipulation trick.

How to Isolate the Last Bit Set

In this example, number x = 1110 in binary:

Binary Digit	1	1	**1**	0
Index	3	2	1	0

The final 1 in the binary digit row is the last bit set, which we want to isolate. But how do we do it?

x & (-x)x& (-x)

This gives us the last bit set in any number, but how?

Let's say that x=a1b in binary, and this number is where we want to isolate the last bit set from.

a is a binary sequence of 1's and 0's (any length), and b is a sequence of 1s (any length). Remember, we only want the last set bit so, for that intermediate 1 bit in between a and b to be that last bit, b must be a sequence of 0's of length zero or more.

-x = 2's complement of x = (a1b)' + 1 = a'0b' + 1 = a'0(0….0)' + 1 = a'0(1…1) + 1 = a'1(0…0) = a'1b

a1b – this is x

& a^{-}1b - this is-x

———-

= (0…0) 1 (0…0) - this is the last set bit isolated

Another example is x = 10(in decimal) = 1010 (in binary)

We get the last bit set by

x&(-x) = (10)1(0) & (01)1(0) = 0010 = 2x&(-x) = (10)1(0) &   01)1(0) = 0010 = 2 (in decimal)

Is there a good reason why we need this last odd bit isolated from any number? Yes, and as we continue through this section, you will see why.

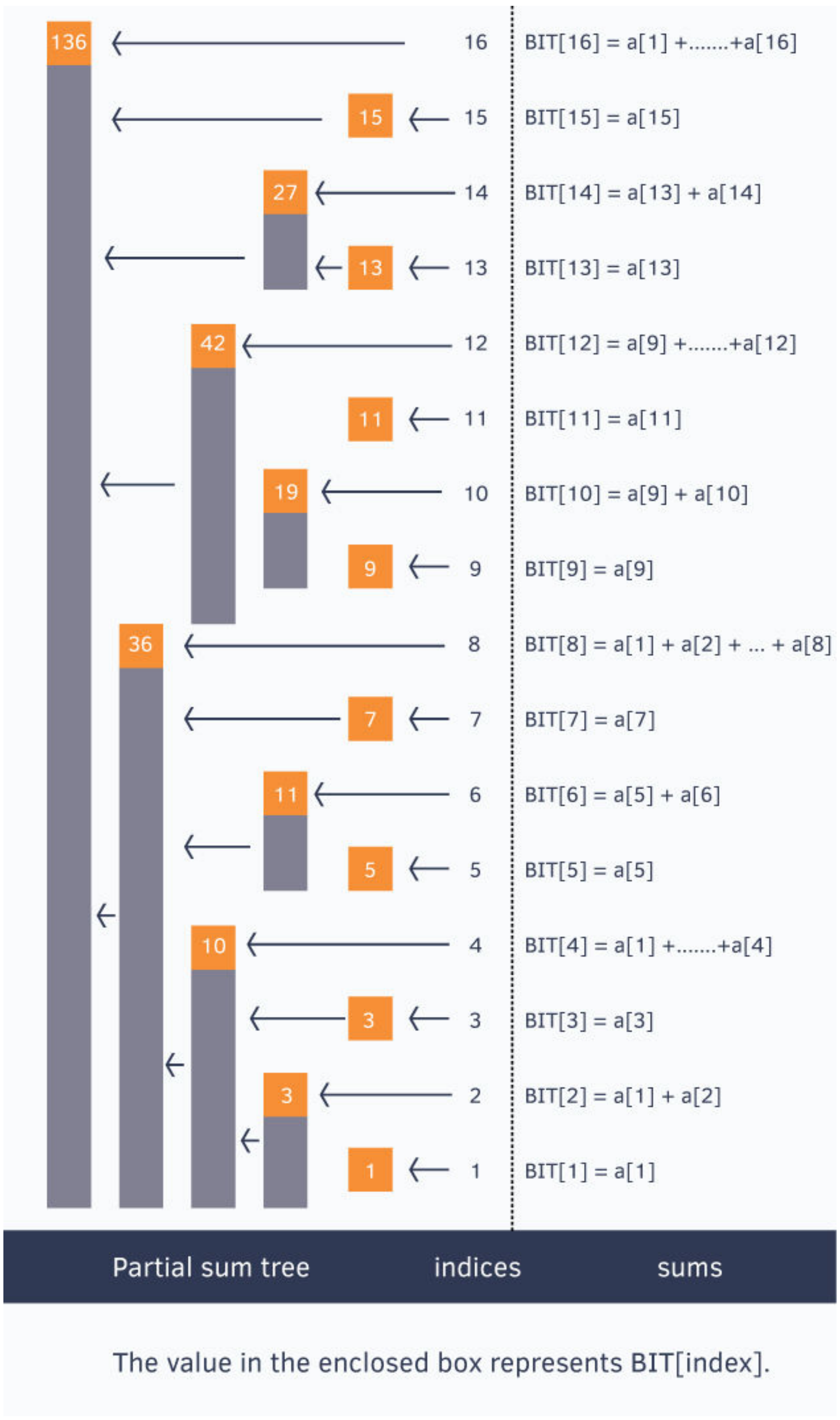
Now let’s look deeper into the BIT tree.

What Is the Idea of a Binary Indexed Tree?

We already know that the sum of powers of two can represent an integer. In the same way, for an array of size N, an array BIT[] can be maintained so that the sum of some of the numbers in the specified array may be stored at any index. This is also known as a partial sum tree.

Here’s an example to show you how partial sums are stored in BIT[].

```
// make sure our given array is 1-based indexed

int a[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16};
```



Courtesy of: https://www.hackerearth.com ›

In the image above, you can see the BIT tree, where the enclosed boxes indicate the value BIT[index], and the partial sum of some numbers is stored in each BIT[index].

Note that:

$$\{ \quad a[x], \quad \text{if } x \text{ is odd}$$

```
BIT[x] =                a[1] + … + a[x],     if x is power of 2

         }
```

The cumulative sum of index i to i-(1<<r)+1 (both inclusive) is stored in each index i in the BIT[] array, with r representing index i's last set bit.

sum of the first 12 numbers in this array a[]=BIT[12]+BIT[8]=(a[12]+…+a[9])+(a[8]+…+a[1])

In the same way

sum of first 6 elements =BIT[6]+BIT[4]=(a[6]+a[5])+(a[4]+…+a[1])

sum of first 8 elements=BIT[8]=a[8]+…+a[1]

Let's construct our tree and then query it for prefix sums.

BIT[] is an array, size = 1 + the given array a[] size, which is where the operations need to be performed. To start with, all the BIT] values are equal to 0.

Next, the update() operation is called for each array element, which constructs the BIT tree. We'll talk about the update operation below.

Let's see how to construct this tree, and then we will come back to querying the tree for prefix sums. BIT[] is an array of size = 1 + the size of the given array a[] on which we need to perform operations. Initially, all values in BIT[] are equal to 0. Then we call update() operation for each element of the given array to construct the Binary Indexed Tree.
The update() operation is discussed below.

```
void update(int x, int val)       //add “val” at index “x”
{
    for(; x <= n; x += x&-x)
         BIT[x] += val;
}
```

Don’t worry if you can’t figure out this update function. I’ll show you an update to help you:

Let’s say we want to call

```
update(13, 2)
```

The above figure shows that index 13 is covered by indices 13, 14, and 16, so we also need to add 2.

x is 13, so BIT[13] is updated:

```
BIT[13] += 2;
```

Now the last bit set of x=13(1101) needs to be isolated and then added to x, for example:

```
x += x&(-x)x += x&(-x)
```

The last bit is 1, and this is added to x:

```
x=3+1=14
```

And then BIT[14] is updated:

```
BIT[14] += 2;
```

14 is now 1110, so we need to isolate the last bit and add it to 14, so x will then become:

```
x=14+2=16(10000)
```

Then, BIT[16] is updated:

```
BIT[16] += 2;
```

When we perform an update operation n index x, all the BIT[] indices are updated, including index x, and BIT[] is maintained.

In the update() operation, you can see a for loop. You can see that it runs the number of bits included in index x, which, as we know, is restricted to be equal to or less than N, which is given array size. So, we could say that the operation would take O(logN) time at the most.

How do we query a structure like this for prefix sums?

Here’s the query operation:

```
int query(int x)      //returns the sum of first x elements in
given array a[]
{
     int sum = 0;
     for(; x > 0; x -= x&-x)
        sum += BIT[x];
     return sum;
}
```

This query will return the sum of the first x number of elements in the array. Here's how it works:

Let's say we call query(14), which is

sum = 0

X is 14(1110), and we add BIT[14] to the sum variable, so

sum=BIT[14]=(a[14]+a[13]

Now, the last set bit is isolated from x=14(1110) and subtracted from x. The last bit in 12(1100) is 4(100), so

x=4-2=12

BIT[12] is added to the sum variable, so

sum=BIT[14]+BIT[12]=(a[14]+a[13])+(a[12]+…+a[9])

Again, the last bit set is isolated from x=12(1100) and subtracted from x. the last bit in 12(1100) is 4(100) so

x=12–4=8

BIT[8] is added to the sum variable, so

sum=BIT[14]+BIT[2]+BIT[8]=(a[14]+a[13])+(a[12]+…+a[9])+(a[8]+…+a[1])

Lastly, the last set bit is isolated from x=8(1000) and subtracted from x. The last bit in (1000) is 8(000), so

x=8–8=0

The for loop will break because x=0 and the prefix sum is returned.

In terms of time complexity, we can see the loop iterating over the number of bits in x, which is N at most. So, it’s safe to say that the query operation will take O(logN) time.

Here’s the whole program:

```
int BIT[1000], a[1000], n;
void update(int x, int val)
{
       for(; x <= n; x += x&-x)
         BIT[x] += val;
}
int query(int x)
{
     int sum = 0;
     for(; x > 0; x -= x&-x)
       sum += BIT[x];
     return sum;
}

int main()
{
     scanf(“%d”, &n);
     int i;
     for(i = 1; i <= n; i++)
     {
          scanf(“%d”, &a[i]);
          update(i, a[i]);
```

```
        }
        printf("sum of first 10 elements is %d\n", query(10));
         printf("sum of all elements in range [2, 7] is %d\n",
    query(7) – query(2-1));
        return 0;
    }
```

When BIT or Fenwick Trees Should be Used

Before you choose to use this tree for performing operations over range, you should first discern whether your function or operation

1. **Is Associative** – i.e. f(f(a,b),c)=f(a,f(b,c)). This applies even for the segment tree.
2. **Has an Inverse** – for example:
   - Addition has an inverse subtraction
   - Multiplication has an inverse division
   - gcd() doesn't have an inverse, so BIT cannot be used to calculate range gcds
   - The sum of the matrices has an inverse
   - The product of the matrices would have an inverse if matrices are given as degenerate, i.e., the matrix determinant does not equal 0
3. **Space Complexity** – is O(N) to declare another size N array
4. **Time Complexity** – is O(logN) for every operation, including query and update

BIT Applications

There are two primary applications for BIT:

1. They are used for implementing the arithmetic coding algorithm. This is the use case that primarily motivated by the development of operations that this supports
2. They are used for counting inversions in arrays in O(NlogN) time

AVL Tree

The AVL tree was first invented in 1962 by GM Adelson-Velsky and EM Landis and was named after the inventors.

The definition of an AVL tree is a height-balanced binary search tree, where every node has a balance factor associated with it. This balance factor is calculated by subtracting the right subtree height from the left subtree height. Provided each node has a balance factor of -1 to +1, the tree is balanced. If not, the tree is unbalanced and needs work to balance it.

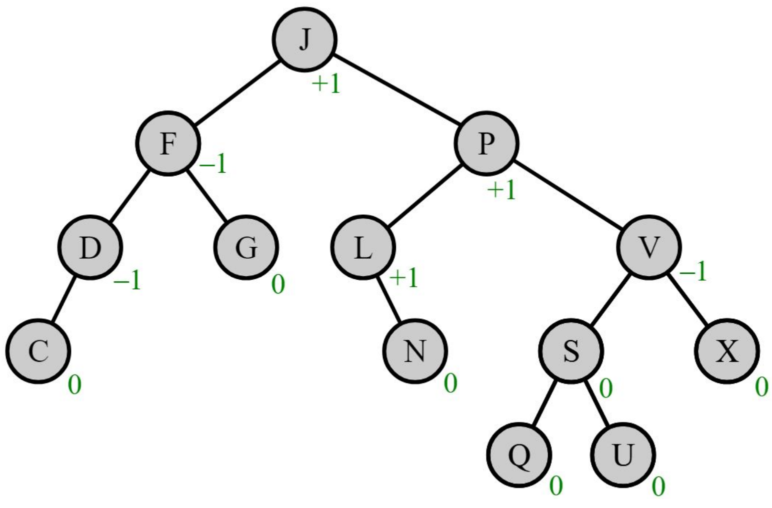
Balance Factor (k) = height (left(k)) - height (right(k))

If any node has a balance factor of 1, the left subtree is higher than the right tree by one level.

If any node has a balance factor of 0, the left and right subtrees are of equal height.

If any node has a balance factor of -1, the left subtree is lower than the right subtree by one level.

Below you can see an illustration of an AVL tree. Each node has a balance factor of -1 to +1:



Complexity

Algorithm	Average Case	Worst Case
Space	O(N)	O(N)
Search	O(logN)	O(logN)
Insert	O(logN)	O(logN)
Delete	O(logN	O(logN)

AVL Tree Operations

Because an AVL tree is a binary search tree, all the operations are performed the same way they are on a standard binary search tree. When you search or traverse an AVL tree, you are not violating the properties but insert and delete operations can violate them.

Operation	**Description**
Insertion	When you perform insertion into an AVL tree, you do it in the same way as any other binary search tree. However, because it can violate the AVL tree property, the tree will need to be rebalanced. This is done by applying rotations, which we will discuss shortly.
Deletion	Deletion is also done the same way as any other binary search tree, but, like the insertion operation, it too can disturb the tree's balance. Again, rotation can help rebalance it.

Why Use an AVL Tree?

AVL trees control the binary search tree height by ensuring it doesn't get skewed or unbalanced. All operations done on a binary search tree of height h take O(h) time. However, should the search tree become skewed, it can take O(N) time.

With the height limited to Logn, the AVL tree places an upper bound on every operation, ensuring it is no more than O(logN), where N indicates how many nodes there are.

Rotations

Rotation is only performed on an AVL tree when the balance factor is not -1, 0, or +1. There are four rotation types:

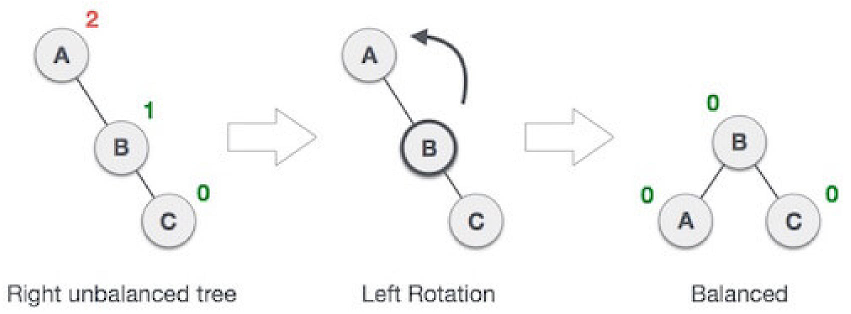
1. **LL Rotation** – the inserted node is in the left subtree of A's left subtree
2. **RR Rotation** – the inserted node is in the right subtree of A's right subtree
3. **LR Rotation** – the inserted node is in the right subtree of A's left subtree
4. **RL Rotation** – the inserted node is in the left subtree of A's right subtree.

In this case, A is the node with the balance factor that isn't between -1 and +1.

LL and RR rotations are single rotations, while LR and RL are double. A tree must be a minimum of 2 in height to be considered unbalanced. Let's look at each rotation:

1. **RR Rotation**

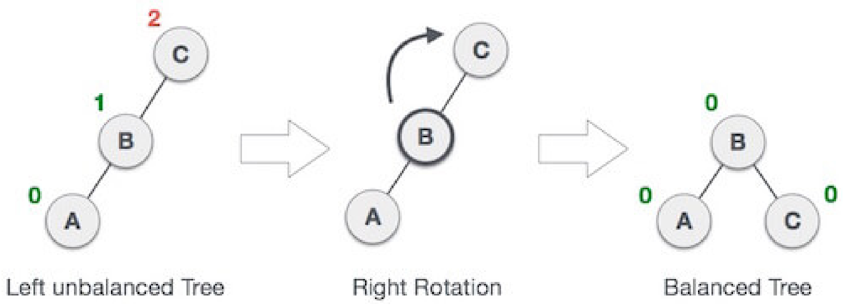
A binary search tree becomes unbalanced when a node has been inserted into the right subtree of A's right subtree. In this case, we can do RR rotation. This rotation is anticlockwise and is applied to the edge below the node with a balance factor of -2:



In this example, node A has a -2 balance factor because Node C has been inserted into the right subtree of A’s right subtree. The RR rotation is performed to the edge beneath A.

2. **LL Rotation**

A binary tree may also become unbalanced when a node has been inserted into the left subtree of C’s left subtree. In this case, we can do LL rotation. This rotation is clockwise, and is applied to the edge underneath the node with a balance factor of 2:



In this example, node C has a 2 balance factor, because node A has been inserted into the left subtree of C’s left subtree. The LL rotation is performed to the edge below A.

3. **LR Rotation**

Double rotations are not so easy as single rotations. To demonstrate, an LR rotation is a combination of an RR and LL rotation, i.e. the RR rotation is done on the subtree and the LL rotation on the full tree – the first node from the inserted node’s path, where the inserted node has a balance factor that is not between -1 and +1.

4. **RL Rotation**

The RL rotation is a comination of an LL and an RR rotation. The LL rotation is done on the subtree and then the RR roation on the full tree, ie., the first node from the inserted node's path, where the inserted node has a balance factor that is not between -1 and +1.

AVL Tree Implementation

The following is a C++ program to show you all the operations on the AVL tree:

```
#include<iostream>
using namespace std;

// An AVL tree node
class AVLNode
{
    public:
    int key;
    AVLNode *left;
    AVLNode *right;
    int depth;
};

//get max of two integers
int max(int a, int b){
    return (a > b)? a : b;
}

//function to get height of the tree
int depth(AVLNode *n)
{
    if (n == NULL)
```

```
        return 0;
    return n->depth;
}
// allocate a new node with key passed
AVLNode* newNode(int key)
{
    AVLNode* node = new AVLNode();
    node->key = key;
    node->left = NULL;
    node->right = NULL;
    node->depth = 1; // new node added as leaf
    return(node);
}
// right rotate the sub tree rooted with y
AVLNode *rightRotate(AVLNode *y)
{
    AVLNode *x = y->left;
    AVLNode *T2 = x->right;

    // Perform rotation
    x->right = y;
    y->left = T2;

    // Update heights
    y->depth = max(depth(y->left),
                   depth(y->right)) + 1;
    x->depth = max(depth(x->left),
                depth(x->right)) + 1;

    // Return new root
    return x;
}

// left rotate the sub tree rooted with x
AVLNode *leftRotate(AVLNode *x)
{
    AVLNode *y = x->right;
    AVLNode *T2 = y->left;

    // Perform rotation
    y->left = x;
    x->right = T2;
    // Update heights
    x->depth = max(depth(x->left),
                depth(x->right)) + 1;
    y->depth = max(depth(y->left),
                depth(y->right)) + 1;

    // Return new root
    return y;
}
// Get Balance factor of node N
int getBalance(AVLNode *N)
{
    if (N == NULL)
        return 0;
    return depth(N->left) -
           depth(N->right);
}
//insertion operation for node in AVL tree
AVLNode* insert(AVLNode* node, int key)  {
    //normal BST rotation
    if (node == NULL)
        return(newNode(key));

    if (key < node->key)
        node->left = insert(node->left, key);
    else if (key > node->key)
        node->right = insert(node->right, key);
  else // Equal keys not allowed
        return node;

    //update height of ancestor node
    node->depth = 1 + max(depth(node->left),
depth(node->right));

    int balance = getBalance(node);        //get balance
factor

    // rotate if unbalanced

    // Left Left Case
    if (balance > 1 && key < node->left->key)
        return rightRotate(node);

    // Right Right Case
    if (balance < -1 && key > node->right->key)
        return leftRotate(node);
  // Left Right Case
    if (balance > 1 && key > node->left->key)
    {
        node->left = leftRotate(node->left);
        return rightRotate(node);
    }

    // Right Left Case
    if (balance < -1 && key < node->right->key)
    {
        node->right = rightRotate(node->right);
        return leftRotate(node);
    }
    return node;
}

// find the node with minimum value
AVLNode * minValueNode(AVLNode* node)
{
    AVLNode* current = node;
    // find the leftmost leaf */
  while (current->left != NULL)
        current = current->left;

    return current;
}
// delete a node from AVL tree with the given key
AVLNode* deleteNode(AVLNode* root, int key)
{
    if (root == NULL)
        return root;

    //perform BST delete
    if ( key < root->key )
        root->left = deleteNode(root->left, key);

    else if( key > root->key )
        root->right = deleteNode(root->right, key);

else
    {
        // node with only one child or no child
        if( (root->left == NULL) ||
            (root->right == NULL) )
        {
            AVLNode *temp = root->left ?
                         root->left :
```

```
                root->right;

          if (temp == NULL)
          {
             temp = root;
             root = NULL;
          }
          else // One child case
          *root = *temp;
          free(temp);
       }
else
       {
          AVLNode* temp = minValueNode(root->right);

          root->key = temp->key;

          // Delete the inorder successor
          root->right = deleteNode(root->right,
                                   temp->key);
       }
   }

   if (root == NULL)
   return root;

   // update depth
```

```
root->depth = 1 + max(depth(root->left),
                 depth(root->right));

// get balance factor
int balance = getBalance(root);

//rotate the tree if unbalanced

// Left Left Case
if (balance > 1 &&
   getBalance(root->left) >= 0)
   return rightRotate(root);

// Left Right Case
if (balance > 1 &&  getBalance(root->left) < 0)  {
   root->left = leftRotate(root->left);
   return rightRotate(root);
}
// Right Right Case
if (balance < -1 &&  getBalance(root->right) <= 0)
   return leftRotate(root);

// Right Left Case
if (balance < -1 && getBalance(root->right) > 0)   {
   root->right = rightRotate(root->right);
   return leftRotate(root);
}
```

```
    return root;
}
// prints inOrder traversal of the AVL tree
void inOrder(AVLNode *root)
{
    if(root != NULL)
    {
        inOrder(root->left);
        cout << root->key << ” “;
        inOrder(root->right);
    }
}
  // main code
int main()
{
    AVLNode *root = NULL;

    // constructing an AVL tree
    root = insert(root, 12);
    root = insert(root, 8);
    root = insert(root, 18);
    root = insert(root, 5);
    root = insert(root, 11);
    root = insert(root, 17);
    root = insert(root, 4);

    //Inorder traversal for above tree : 4 5 8 11 12 17 18
    cout << “Inorder traversal for the AVL tree is: \n”;
    inOrder(root);
    root = deleteNode(root, 5);
    cout << “\nInorder traversal after deletion of node 5: 
\n”;
    inOrder(root);


    return 0;
}
```

Output :

```
Inorder traversal for the AVL tree is:
4 5 8 11 12 17 18
Inorder traversal after deletion of node 5:
4 8 11 12 17 18
```

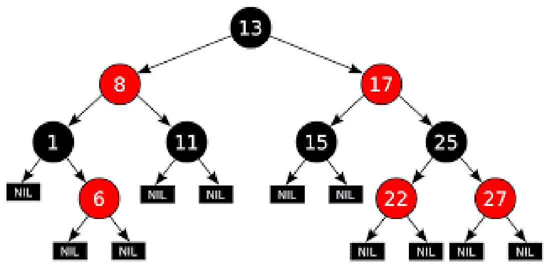
Red Black Tree

Red-black trees are also a form of a self-balancing binary tree, created by Rudolf Bayer in 1972. In this type of binary tree, each node is given an extra attribute – a color, red or black. The tree checks the node colors on the path between the root to a leaf, ensuring that no path is any more than twice the length of any other. That way, the tree is balanced.

Red-Black Tree Properties

Red-black trees must satisfy several properties:

- The root must always be black
- NILs are recognized as black – all non-NIL nodes will have two children.
- Red node children are always black
- The black height rule dictates that, for a specific node v, there is an integer bh(v) in such a way that a specific path down to a NIL from v has the correct bh(v) real nodes. The black height of a red-black tree is determined as the root's black height.

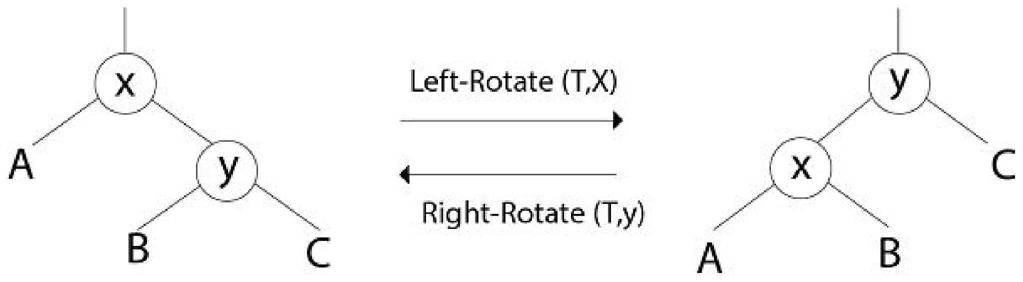


Red-Black tree Operations

These are very similar to those on an AVL tree and include Delete and Insert. When these are run on a tree with n keys, they take O(logN) time. However, these operations customize the red-black tree and could violate the properties. These properties can be restored by changing the colors on some of the tree nodes and changing the pointer structure.

1. **Rotation**

The rotation operation clearly expresses the restructuring operations on these trees:



The rotation operation preserves the Ax by C order. So, if we begin with a binary search tree and only use rotation to restructure, we will keep the binary search tree – rotation doesn't break the tree's property.

LEFT ROTATE (T, x)

```
y ← right [x]
y ← right [x]
right [x] ← left [y]
p [left[y]] ← x
p[y] ← p[x]
If p[x] = nil [T]
    then root [T] ← y
      else if x = left [p[x]]


        then left [p[x]] ← y
      else right [p[x]] ← y
left [y] ← x.
p [x] ← y.
```

2. **Insertion**

New nodes are inserted in the same way they are in binary search trees, and the node is colored red. If there are any discrepancies in the tree, they must be fixed according to the discrepancy type.

Discrepancies can arise from parent and child nodes both being red and the node location determines this in relation to the grandparent and the parent's sibling's color.

RB-INSERT (T, z)

```
y ← nil [T]
x ← root [T]
while x ≠ NIL [T]
do y ← x
if key [z] < key [x]
then x  ← left [x]
else x ←  right [x]
p [z] ← y
if y = nil [T]
then root [T] ← z
else if key [z] < key [y]
then left [y] ← z
else right [y] ← z
left [z] ← nil [T]
right [z] ← nil [T]
color [z] ← RED
RB-INSERT-FIXUP (T, z)
```

After a new node has been inserted, coloring it black could violate the conditions set for black height. By the same token, coloring it red could violate the conditions set for coloring, i.e., the root should be black, and rede nodes do not have any children.

We know that violations of the black height rules are hard, so it's best to color the node red. After, if a color violation arises, it must be fixed using RB-INSERT-FIXUP:

RB-INSERT-FIXUP (T, z)

```
while color [p[z]] = RED
do if p [z] = left [p[p[z]]]
then y ← right [p[p[z]]]
If color [y] = RED
then color [p[z]] ← BLACK    //Case 1
color [y] ← BLACK            //Case 1
color [p[z]] ← RED           //Case 1
z  ← p[p[z]]                 //Case 1
else if z= right [p[z]]
then z ← p [z]               //Case 2
LEFT-ROTATE (T, z)           //Case 2
color [p[z]] ← BLACK         //Case 3
color [p [p[z]]] ← RED       //Case 3
RIGHT-ROTATE  (T,p [p[z]])  //Case 3
else (same as then clause)
  With "right" and "left" exchanged
color [root[T]] ← BLACK
```

3. **Deletion**

Node deletion requires several steps:

- First, you must find the element you want to delete
- If it is in a node with only a left child, the node must be swapped with another in the left subtree with the biggest element. This node will not have a right child.
- If it is in a node with only a right child, the node must be swapped with another in the right subtree with the smallest element. This node will not have a left child.
- If it is in a node with a left and right child, it can be swapped in either of the above two ways. While swapping, ensure only the keys are swapped, not the colors.

- The element you want to be deleted now has a left child OR a right child, so the node can be replaced with the solo child. However, be aware that this could result in a violation of the red or black color constraints.
- If the node you delete is black, it violates the back constraint. Eliminating a black node y will result in any path containing y having one less black node.
- This can result in one of two cases arising:
  - The replacement node is red, so we can just color it in black, thus eliminating the loss of a black node
  - The replacement node is black.

The RB-DELETE strategy is a small deviation from the TREE-DELETE strategy. Once a node has been spliced, an auxiliary procedure, RB-DELETE-FIXUP, is called to change the colors and restore the red-black properties by performing a rotation.

RB-DELETE (T, z)

```
    if left [z] = nil [T] or right [z] = nil [T]
    then y ← z
    else y ← TREE-SUCCESSOR (z)
    if left [y] ≠ nil [T]
    then x ← left [y]
    else x ← right [y]
    p [x] ←  p [y]
    if p[y] = nil [T]
    then root [T]  ← x
    else if y = left [p[y]]
    then left [p[y]] ← x
    else right [p[y]] ← x
    if y≠ z
    then key [z] ← key [y]
```

```
        copy y’s satellite data into z
        if color [y] = BLACK
        then RB-delete-FIXUP (T, x)
        return y
```

RB-DELETE-FIXUP (T, x)

```
        while x ≠ root [T] and color [x] = BLACK
        do if x = left [p[x]]
        then w ← right [p[x]]
        if color [w] = RED
        then color [w] ← BLACK        //Case 1
        color [p[x]] ← RED            //Case 1
        LEFT-ROTATE (T, p [x])        //Case 1
        w ← right [p[x]]              //Case 1
        If color [left [w]] = BLACK and color [right[w]] =
        BLACK
        then color [w] ← RED          //Case 2
        x ← p[x]                      //Case 2
        else if color [right [w]] = BLACK
        then color [left[w]] ← BLACK //Case 3
        color [w] ← RED               //Case 3
        RIGHT-ROTATE (T, w)           //Case 3
        w ← right [p[x]]              //Case 3
        color [w] ← color [p[x]]      //Case 4
        color p[x] ← BLACK            //Case 4
        color [right [w]] ← BLACK     //Case 4
        LEFT-ROTATE (T, p [x])        //Case 4
        x ← root [T]                  //Case 4
```

```
else (same as then clause with "right" and "left" exchanged)
  color [x] ← BLACK
```

Why Use a Red-Black Tree?

Most binary search tree operations, for example, min, max, search, delete, insert, etc., take O(H) time, with h being the tree height. If the binary tree becomes skewed, that cost can rise to O(N). Provided we ensure the tree height stays as O(logN) after each insert and delete action, an upper bound is guaranteed for every operation of O(logN). A red-black tree always has a height of O(logN), where N indicates how many nodes are in the tree.

Interesting Points:

1. The black height indicates how many black nodes are on the path from the root to the leaf node. Leaf nodes are black nodes, too, so if a red-black tree with a height of h has a black height of >=h/2.
2. Where the red-black tree has n nodes, the height is $h<=2 \log_2(n + 1)$.
3. All leaf nodes are NIL and black
4. A node's back depth is defined by how many black nodes there are between the root and the node, i.e., how many black ancestors are present.
5. All red-black trees are variations on the binary search tree.

Black Height

So, we know that the black height is indicated by how many black nodes are present on the path between the root and the node, and we also know that leaf nodes are considered black nodes. From the third and fourth properties above, we can determine that a red-black tree with a height of h is black-height >=h/2. Keep in mind that the number of nides between a node and its furthest descendent (leaf) cannot be any more than twice the number to the nearest descendent.

All red-black trees with n nodes will have a height of $<=2Log_2(n + 1)$, and we can prove that with the following:

1. In a standard binary tree, let's say that k is the minimum number of nodes on all paths between the root and NULL. In that case, $n>=2^k-1$. For example, if k were 3, n would be 7 as a minimum. We can also write this same expression as $k <= Log_2(n+1)$.
2. From the fourth property above, we could say that, in a red-black tree with n nodes, a root to leaf path has a minimum of $Log_2(n+1)$ black nodes.
3. From the third property, we could say that the number of the black nodes in the tree is a minimum of | n/2 |, where n indicates how many nodes there are in total.

From this, we can draw the conclusion that a red-black tree with n nodes is of height $<=2Log_2(n+1)$.

Here's a Java code implementation of red-black tree traversal and insertion:

```
/*package whatever //do not write package name here */


import java.io.*;


// RedBlackTree class. This class contains the subclass
for the node

// and the functionalities of RedBlackTree such as -
rotations, insertion and

// in-order traversal

public class RedBlackTree

{

    public Node root;//root node

    public RedBlackTree()

    {
        super();
        root = null;
    }
    // node creating subclass
    class Node
    {
        int data;
        Node left;
        Node right;
        char color;
        Node parent;

        Node(int data)
        {
            super();
            this.data = data;   // only including data. not key
            this.left = null; // left subtree
            this.right = null; // right subtree
            this.color = ‘R’; // color . either ‘R’ or ‘B’
           this.parent = null; // required at time of
rechecking.
        }
    }
    // this function performs left rotation
    Node rotateLeft(Node node)
    {
        Node x = node.right;
    Node y = x.left;
    x.left = node;
    node.right = y;
    node.parent = x; // parent resetting is also important.
    if(y!=null)
        y.parent = node;
    return(x);
}
//this function performs right rotation
Node rotateRight(Node node)
{
    Node x = node.left;
    Node y = x.right;
    x.right = node;
    node.left = y;
    node.parent = x;
    if(y!=null)
        y.parent = node;
    return(x);
}


// these are some flags.
// Respective rotations are performed during traceback.
// rotations are done if flags are true.
boolean ll = false;
boolean rr = false;
```

```
    boolean lr = false;
    boolean rl = false;
    // helper function for insertion. Actually this function
performs all tasks in single pass only.
    Node insertHelp(Node root, int data)
    {
        // f is true when RED conflict is there.
        boolean f=false;

        //recursive calls to insert at proper position
according to BST properties.
        if(root==null)
            return(new Node(data));
        else if(data<root.data)
        {
            root.left = insertHelp(root.left, data);
            root.left.parent = root;
            if(root!=this.root)
            {
                if(root.color=='R' && root.left.color=='R')
                    f = true;
            }
        }
        else
        {
            root.right = insertHelp(root.right,data);
            root.right.parent = root;
            if(root!=this.root)
            {
                if(root.color==‘R’ && root.right.color==‘R’)
                    f = true;
            }
        // at the time of insertion, we also assign parent
nodes
        // and check for RED RED conflicts
        }

        // now let’s rotate.
        if(this.ll) // for left rotate.
        {
            root = rotateLeft(root);
            root.color = ‘B’;
            root.left.color = ‘R’;
            this.ll = false;
        }
        else if(this.rr) // for right rotate
        {
            root = rotateRight(root);
            root.color = ‘B’;
            root.right.color = ‘R’;
            this.rr  = false;
        }
        else if(this.rl)  // for right and then left
        {
            root.right = rotateRight(root.right);
            root.right.parent = root;
            root = rotateLeft(root);
            root.color = ‘B’;
            root.left.color = ‘R’;

            this.rl = false;
        }
        else if(this.lr)  // for left and then right.
        {
            root.left = rotateLeft(root.left);
            root.left.parent = root;
            root = rotateRight(root);
            root.color = ‘B’;
            root.right.color = ‘R’;
            this.lr = false;
        }
        // when rotation and recoloring are finished, the
flags are reset.
        // Now take care of the RED RED conflict
        if(f)
        {
            if(root.parent.right == root)  // to check which
child is the current node of its parent
            {
                if(root.parent.left==null ||
root.parent.left.color==‘B’)  // case when parent’s sibling
is black
                {// perform rotation and recoloring while
backtracking, i.e. setting up respective flags.
```

```
                if(root.left!=null && root.left.color=='R')
                    this.rl = true;
                else if(root.right!=null &&
root.right.color=='R')
                    this.ll = true;
            }
            else // case when parent's sibling is red
            {
                root.parent.left.color = 'B';
                root.color = 'B';
                if(root.parent!=this.root)
                    root.parent.color = 'R';
            }
        }
        else
        {
            if(root.parent.right==null ||
root.parent.right.color=='B')
            {
                if(root.left!=null && root.left.color=='R')
                    this.rr = true;
                else if(root.right!=null &&
root.right.color=='R')
                    this.lr = true;
            }
            else
            {
                root.parent.right.color = 'B';
                root.color = ‘B’;
                if(root.parent!=this.root)
                    root.parent.color = ‘R’;
            }
        }
        f = false;
    }
    return(root);
}

// function to insert data into tree.
public void insert(int data)
{
    if(this.root==null)
    {
        this.root = new Node(data);
        this.root.color = ‘B’;
    }
   else
        this.root = insertHelp(this.root,data);
}
// helper function to print inorder traversal
void inorderTraversalHelper(Node node)
{
    if(node!=null)
    {
        inorderTraversalHelper(node.left);
        System.out.printf("%d ", node.data);
        inorderTraversalHelper(node.right);
    }
}
//function to print inorder traversal
public void inorderTraversal()
{
    inorderTraversalHelper(this.root);
}
// helper function to print the tree.
void printTreeHelper(Node root, int space)
{
    int i;
    if(root != null)
    {
        space = space + 10;
        printTreeHelper(root.right, space);
        System.out.printf("\n");
        for ( i = 10; i < space; i++)
        {
            System.out.printf(" ");
        }
        System.out.printf("%d", root.data);
        System.out.printf("\n");
        printTreeHelper(root.left, space);
    }
}
```

```
    // function to print the tree.
    public void printTree()
    {
        printTreeHelper(this.root, 0);
    }
    public static void main(String[] args)
    {
        // try inserting some data into tree and visualizing
the tree as well as traversing it.
        RedBlackTree t = new RedBlackTree();
        int[] arr = {1,4,6,3,5,7,8,2,9};
        for(int i=0;i<9;i++)
        {
            t.insert(arr[i]);
            System.out.println();
            t.inorderTraversal();
        }
        // check the color of any node by its attribute
node.color
        t.printTree();
    }
}
```

The output is:

```
    1
    1 4
    1 4 6
    1 3 4 6
```

1 3 4 5 6

1 3 4 5 6 7

1 3 4 5 6 7 8

1 2 3 4 5 6 7 8

1 2 3 4 5 6 7 8 9

9

8

7

6

5

4

3

2

1

Scapegoat Trees

Scapegoat trees are another variant of the self-balancing binary search tree. However, unlike others, such as the AVL and red-black trees, scapegoats do not need any extra space per node for storage. They are easy to implement and have low overhead, making them one of the most attractive data structures. They are generally used in applications where lookup and insert operations are dominant, as this is where their efficiency lies.

The idea behind a scapegoat tree is based on something we have all dealt with at one time or another – when something goes wrong, we look for a scapegoat to blame it on. Once the scapegoat is identified, we leave them to deal with what went wrong. After a nide is inserted, the scapegoat will find a node with an unbalanced subtree – that is your scapegoat. The subtree will then be rebalanced.

In terms of implementation, a scapegoat tree is flexible. You can optimize them for insert operations at the expense of a lookup or delete operation, or vice-versa. The programmer has carte blanche in tailoring the tree to the application, making these data structures attractive to most programmers.

Finding a scapegoat for insertion is a simple process. Let's say we have a balanced tree. A new node is inserted, subsequently unbalancing the tree. Starting at the node we just inserted, we look at the node's subtree root – if it is balanced, we move on to the parent node and look again. Somewhere along the line, we will discover a scapegoat with an unbalanced subtree – this is what must be fixed to rebalance the tree.

Properties

Out of all the binary search trees, the scapegoat is the first to achieve its complexity of O(\log_2(n))O(log2 (n)) without needing additional information stored at every node. This represents huge savings in terms of space, making it one of the best data structures to use when space is short.

The scapegoat tree stores the following properties for the whole tree:

Property	**Function**
size	Indicates how many nodes are in the tree.
root	A pointer to the tree's root.
max_size	Indicates the maximum tree size since the last total rebuild.

Each tree node has these properties:

Property	**Function**
key	Indicates the node's value
left	A pointer to the node's left child.
right	A pointer to the node's right child.

Operations

The scapegoat tree handles two operations – insertion and deletion – in a unique way, while traversal and lookup are done in the same way as any balanced binary search tree.

Insertion

The insertion process begins in the same way as a binary search tree. A binary search is used to find the right place for the new node, a process a time complexity of O(\log_2(n))O(log2 (n)) time.

Next, the tree must determine if rebalancing is required, so it traverses up the ancestry for the new node, looking for the first node with an unbalanced subtree. A factor of \alphaα, which is the tree's weighing property, is whether the tree is or isn't balanced. For the sake of simplicity, the number of nodes present in both the right and left subtrees cannot be different by more than 1. If the tree is unbalanced, it's a surefire thing that one of the new node's ancestors is unbalanced, common sense considering this is where the new node was added to the balanced tree. The time complexity to traverse up the tree to find the scapegoat is O(\log_2(n))O(log2 (n)), while the time to actually rebalance a tree that is rooted at the scapegoat is O(n)O(n).

However, this may not be a reasonable analysis because, although a scapegoat might be the tree root, it could also be a node deeply buried in the tree. In that case, it would take much less time because the majority of the tree is left untouched. As such, we can say that the amortized time for rebalancing is O(\log_2(n))O(log2 (n)).

Deletion

Deleting nodes from a scapegoat tree is far easier than inserting them. The tree uses the max_size property, indicating the maximum size the tree achieved since it was last fully rebalanced. Whenever a full rebalance is carried out, the scapegoat tree will set max_size to size.

When a node is deleted, the tree looks at the size property to see if it is equal to or less than max_size. If yes, the tree will rebalance itself from the root, taking O(n)O(n) time. However, the amortized time is similar to insertion, O(\log_2(n))O(log2 (n)).

Complexity

Scapegoat tree complexity is similar to other binary search trees

	Average

Space**	O(n)O(n)
Search	O(\log_2(n))O(log2 (n))
Traversal	*O(n)O(n)
Insert	*O(\log_2(n))O(log2 (n))
Delete	*O(\log_2(n))O(log2 (n))

*amortized analysis

** better than other self-balancing binary search trees

Python Implementation

Below is an implementation of the scapegoat tree in Python, showing you the insert operation. Note that all the nodes are pointing to their parent nodes as a way to make the code easier to read. However, you can omit this by tracking every parent node as you traverse down the tree for stack insertion.

The code includes two classes describing the Scapegoat tree and the Node, with their operations:

```
import math

class Node:

    def __init__(self, key):

        self.key = key

        self.left = None

        self.right = None

        self.parent = None


class Scapegoat:

    def __init__(self):
```

```
    self.root = None
    self.size = 0
    self.maxSize = 0

def insert(self, key):
    node = Node(key)
    #Base Case - Nothing in the tree
    if self.root == None:
        self.root = node
        return
    #Search to find the correct place for the node
    currentNode = self.root
    while currentNode != None:
        potentialParent = currentNode
        if node.key < currentNode.key:
            currentNode = currentNode.left
        else:
            currentNode = currentNode.right
    #Assign the new node with parents and siblings
    node.parent = potentialParent
    if node.key < node.parent.key:
        node.parent.left = node
    else:
        node.parent.right = node
    node.left = None
    node.right = None
    self.size += 1
```

```
        scapegoat = self.findScapegoat(node)
        if scapegoat == None:
            return
        tmp = self.rebalance(scapegoat)

        #Assign the right pointers to and from the scapegoat
        scapegoat.left = tmp.left
        scapegoat.right = tmp.right
        scapegoat.key = tmp.key
        scapegoat.left.parent = scapegoat
        scapegoat.right.parent = scapegoat

    def findScapegoat(self, node):
        if node == self.root:
            return None
        while self.isBalancedAtNode(node) == True:
            if node == self.root:
                return None
            node = node.parent
        return node

    def isBalancedAtNode(self, node):
        if abs(self.sizeOfSubtree(node.left) -
self.sizeOfSubtree(node.right)) <= 1:
            return True
        return False
    def sizeOfSubtree(self, node):
        if node == None:
            return 0
        return 1 + self.sizeOfSubtree(node.left) +
self.sizeOfSubtree(node.right)

    def rebalance(self, root):
        def flatten(node, nodes):
            if node == None:
                return
            flatten(node.left, nodes)
            nodes.append(node)
            flatten(node.right, nodes)

        def buildTreeFromSortedList(nodes, start, end):
            if start > end:
                return None
            mid = int(math.ceil(start + (end - start) / 2.0))
            node = Node(nodes[mid].key)
            node.left = buildTreeFromSortedList(nodes, start,
mid-1)
            node.right = buildTreeFromSortedList(nodes,
mid+1, end)
            return node

        nodes = []
        flatten(root, nodes)
        return buildTreeFromSortedList(nodes, 0,
    len(nodes)-1)


      def delete(self, key):
        pass
```

The properties for both classes are in the class constructors. Plus, on line 6, the parent pointer is included to make method writing easier.

On lines 59 and 64, you can see the functions isBalancedAtNode and sizeOfSubtree. These functions inform the insertion function of the scapegoat's location. The function, sizeOfSubtree, searches recursively down the left and right paths of the node, looking to see the number of nodes beneath the node. The function, isBalancedAtTree, uses the information gained from the first function to identify if a rooted subtree at a specified node is or isn't balanced.

On line 15, the insert function is the interesting part. Here, the new node is placed in the right location at the bottom of the tree. From there, we backtrack to find the scapegoat by following the parent pointers. The findScapegoat function on line 50 takes care of this. The while loop in this function (line 53) fails because there is no balanced subtree beneath the node being inspected, so this tree needs to be rebalanced.

You can see the function to rebalance this tree on line 69. First, the tree has to be flattened recursively into a sorted list. Once that is done, a binary search can be performed on the list to create a new, balanced tree. Then the new tree is hooked back into the original tree using the code you can see on lines 44 to 48.

While we didn't include the delete function here, the logic is the same. In fact, deletion is easier because you do not need to search for the scapegoat – it will always be the root node.

Treap

Also a binary search tree, a treap is a little different. It does not come with a height guarantee of O(logN), and maintaining the tree's balance with a high probability is done using the binary heap and randomization properties. The insert, delete, and search functions have an expected time complexity of O(logN).

Each node in a treat will maintain two values:

- **Key** – follow the ordering in a standard binary search tree – the right is greater, and the left is smaller
- **Priority** – a value assigned randomly following the property called Max-Heap.

Basic Operations

Like all other self-balancing trees, the treap maintains the Max-Heap property during the deletion and insertion functions by using rotation.

In the example below, T1, T2, and T3 are all subtrees of the tree on the left side rooted with y or the tree on the right side rooted with x:

```
        y                         x
       / \    Right Rotation     / \
      x   T3  – – – – – – – >   T1   y
     / \      < - - - - - - -       / \
    T1  T2     Left Rotation       T2  T3
```

In both of these trees, the keys follow the order below:

keys(T1) < key(x) < keys(T2) < key(y) < keys(T3)

As you can see, the binary search tree property has not been violated in any way.

Search

Searching a treap tree is the same as searching any standard binary search tree, so no explanations are needed here – you already know how to do that!

Insert

Inserting a new node is done with these steps:

1. A new node is created with key = x and value = a random value.
2. A standard binary search tree insert procedure is then followed.
3. Rotations are performed to ensure that the priority for the newly inserted node follows the Max-Heap property.

Delete

Here are the steps to do this:

1. First, see if the node you want to delete is a leaf – if yes, delete it
2. If the node is not a leaf, the priority must be replaced with -INF (minus infinite), and then the node is brought down to a leaf by performing the right rotations.

Implementing the Operations:

```
// C function that searches a specified key in a specified
binary search tree

TreapNode* search(TreapNode* root, int key)

{

    // Base Cases: the root is null or the key is present at
root

    if (root == NULL || root->key == key)

       return root;


    // The key is greater than the root's key

    if (root->key < key)

       return search(root->right, key);
```

```
        // The key is smaller than the root's key
        return search(root->left, key);
    }
```

Insert

1. The new node is created with key = x and the value = to a random value
2. A standard binary search tree insert is performed
3. When you insert a new node, it is given a random priority, which may violate the Max-Heap property. To ensure priority follows Max-Heap, do the right rotations.
4. When a node is inserted, all of its ancestors are traversed recursively:
    a. If you insert the node in the left subtree and the left subtree's root has priority, a right rotation is needed
    b. If you insert the node in the right subtree and the right subtree's root has priority, a left rotation is needed.

Below you can see an insertion operation in treap implemented recursively:

```
TreapNode* insert(TreapNode* root, int key)
{
    // If the root is NULL, create a new node and return it
    if (!root)
        return newNode(key);


    // If the key is smaller than the root
    if (key <= root->key)
    {
        // Insert in the left subtree
        root->left = insert(root->left, key);
```

```
            // Fix the Heap property if it is violated
            if (root->left->priority > root->priority)
                root = rightRotate(root);
        }
          else  // If the key is greater
          {
            // Insert in the right subtree
            root->right = insert(root->right, key);

            // Fix the Heap property if it is violated
            if (root->right->priority > root->priority)
                root = leftRotate(root);
          }
          return root;
      }
```

Delete

This implementation is a little different from before:

1. Determine if the node you want to be deleted is a leaf – if yes, delete it.
2. If the node has a NULL and a NON-NULL child, the node must be replaced with the NON-NULL child
3. If both of the node's children are NON-NULL, you must find the left and right childrens' MAX:
   a. If the right child's priority is greater, a left rotation is performed at the node
   b. If the left child's priority is greater, a right rotation is performed at the node

The idea behind the third step is to get the node moved down to end up with an a or b situation.

Below you can see a recursive implementation of the Delete operation:

```
  TreapNode* deleteNode(TreapNode* root, int key)
      {
          // Base case
          if (root == NULL) return root;
```

```
// IF THE KEY IS NOT AT THE ROOT
if (key < root->key)
   root->left = deleteNode(root->left, key);
else if (key > root->key)
   root->right = deleteNode(root->right, key);

// IF THE KEY IS AT THE ROOT

// If theleft is NULL
else if (root->left == NULL)
{
   TreapNode *temp = root->right;
   delete(root);
   root = temp;  // Make right child as root
}

// If the Right is NULL
else if (root->right == NULL)
{
   TreapNode *temp = root->left;
   delete(root);
   root = temp;  // Make left child as root
}

// If the key is at the root and both the left and right are not
NULL
```

```
    else if (root->left->priority < root->right->priority)
    {
        root = leftRotate(root);
        root->left = deleteNode(root->left, key);
    }
    else
    {
        root = rightRotate(root);
        root->right = deleteNode(root->right, key);
    }

    return root;
}
```

Here is the complete program in C++ to show all the operations and the output:

```
// C++ program demonstrating search, insert and delete in
Treap
#include <bits/stdc++.h>
using namespace std;

// A Treap Node
struct TreapNode
{
    int key, priority;
    TreapNode *left, *right;
};

/* T1, T2 and T3 are the subtrees of the tree rooted with y
  (on left side) or x (on right side)
```

```
          y                         x
         / \    Right Rotation        /  \
        x   T3   – – – – – – – >       T1   y
       / \      < - - - - - - -          / \
      T1  T2     Left Rotation           T2  T3 */

// A utility function to rotate the subtree rooted with y to
the right
// See the above diagram.
TreapNode *rightRotate(TreapNode *y)
{
    TreapNode *x = y->left,  *T2 = x->right;

    // Perform rotation
    x->right = y;
    y->left = T2;

    // Return the new root
    return x;
}

// A utility function to rotate the subtree rooted with x to
the left
// See the above diagram
TreapNode *leftRotate(TreapNode *x)
{
    TreapNode *y = x->right, *T2 = y->left;
    // Perform the rotation
    y->left = x;
    x->right = T2;

    // Return the new root
    return y;
}

/* Utility function to add a new key */
TreapNode* newNode(int key)
{
    TreapNode* temp = new TreapNode;
    temp->key = key;
    temp->priority = rand()%100;
    temp->left = temp->right = NULL;
    return temp;
}

// C function to search a specified key in a specified
binary search tree
TreapNode* search(TreapNode* root, int key)
{
    // Base Cases: the root is null or key is present at
theroot
    if (root == NULL || root->key == key)
      return root;

    // The key is greater than the root’s key
    if (root->key < key)
       return search(root->right, key);

    // The key is smaller than the root's key
    return search(root->left, key);
}

/* Recursive implementation of insertion in Treap */
TreapNode* insert(TreapNode* root, int key)
{
    // If the root is NULL, create a new node and return it
    if (!root)
       return newNode(key);

    // If the key is smaller than the root
    if (key <= root->key)
    {
        // Insert in the left subtree
        root->left = insert(root->left, key);

        // Fix the Heap property if it is violated
        if (root->left->priority > root->priority)
            root = rightRotate(root);
    }
    else  // If the key is greater
    {
        // Insert in the right subtree
        root->right = insert(root->right, key);

        // Fix the Heap property if it is violated
        if (root->right->priority > root->priority)
            root = leftRotate(root);
    }
    return root;
}

/* Recursive implementation of Delete() */
TreapNode* deleteNode(TreapNode* root, int key)
{
    if (root == NULL)
        return root;

    if (key < root->key)
        root->left = deleteNode(root->left, key);
    else if (key > root->key)
        root->right = deleteNode(root->right, key);

    // IF THE KEY IS AT THE ROOT

    // If the left is NULL
    else if (root->left == NULL)
    {
        TreapNode *temp = root->right;
        delete(root);
        root = temp;  // Make the right child as root
    }

    // If the Right is NULL
    else if (root->right == NULL)
    {
        TreapNode *temp = root->left;
        delete(root);
        root = temp;  // Make the left child as root
    }

    // If the key is at the root and both the left and right are
not NULL
    else if (root->left->priority < root->right->priority)
    {
        root = leftRotate(root);
        root->left = deleteNode(root->left, key);
    }
    else
    {
        root = rightRotate(root);
        root->right = deleteNode(root->right, key);
    }

    return root;
}
```

```
// A utility function to print the tree
void inorder(TreapNode* root)
{
    if (root)
    {
        inorder(root->left);
        cout << “key: “<< root->key << ” | priority: %d “
            << root->priority;
        if (root->left)
            cout << ” | left child: ” << root->left->key;
        if (root->right)
            cout << ” | right child: ” << root->right->key;
        cout << endl;
        inorder(root->right);
    }
}


// Driver Program to test these functions
int main()
{
    srand(time(NULL));

    struct TreapNode *root = NULL;
    root = insert(root, 50);
    root = insert(root, 30);
    root = insert(root, 20);
    root = insert(root, 40);
    root = insert(root, 70);
    root = insert(root, 60);
    root = insert(root, 80);

    cout << “Inorder traversal of the specified tree \n”;
    inorder(root);

    cout << “\nDelete 20\n”;
    root = deleteNode(root, 20);
    cout << “Inorder traversal of the newly modified tree
\n”;
    inorder(root);

    cout << “\nDelete 30\n”;
    root = deleteNode(root, 30);
    cout << “Inorder traversal of the newly modified tree
\n”;
    inorder(root);

    cout << “\nDelete 50\n”;
    root = deleteNode(root, 50);
    cout << “Inorder traversal of the newly modified tree
\n”;
    inorder(root);

    TreapNode *res = search(root, 50);
    (res == NULL)? cout << “\n50 Not Found “:
```

```
                    cout << “\n50 found”;

          return 0;
     }
```

Output:

```
     Inorder traversal of the specified tree
     key: 20 | priority: %d 92 | right child: 50
     key: 30 | priority: %d 48 | right child: 40
     key: 40 | priority: %d 21
     key: 50 | priority: %d 73 | left child: 30 | right child: 60
     key: 60 | priority: %d 55 | right child: 70
     key: 70 | priority: %d 50 | right child: 80
     key: 80 | priority: %d 44

     Delete 20
     Inorder traversal of the newly modified tree
     key: 30 | priority: %d 48 | right child: 40
     key: 40 | priority: %d 21
     key: 50 | priority: %d 73 | left child: 30 | right child: 60
     key: 60 | priority: %d 55 | right child: 70
     key: 70 | priority: %d 50 | right child: 80
     key: 80 | priority: %d 44

     Delete 30
     Inorder traversal of the newly modified tree
     key: 40 | priority: %d 21
     key: 50 | priority: %d 73 | left child: 40 | right child: 60
```

```
key: 60 | priority: %d 55 | right child: 70
key: 70 | priority: %d 50 | right child: 80
key: 80 | priority: %d 44

Delete 50
Inorder traversal of the newly modified tree
key: 40 | priority: %d 21
key: 60 | priority: %d 55 | left child: 40 | right child: 70
key: 70 | priority: %d 50 | right child: 80
key: 80 | priority: %d 44

50 Not Found
```

An Explanation of the Output:

All the nodes are written as key(priority), and the code above will give us this tree:

```
 20(92)
    \
    50(73)
    /    \
 30(48)   60(55)
    \       \
 40(21)     70(50)
              \
              80(44)
```

After deleteNode(20)

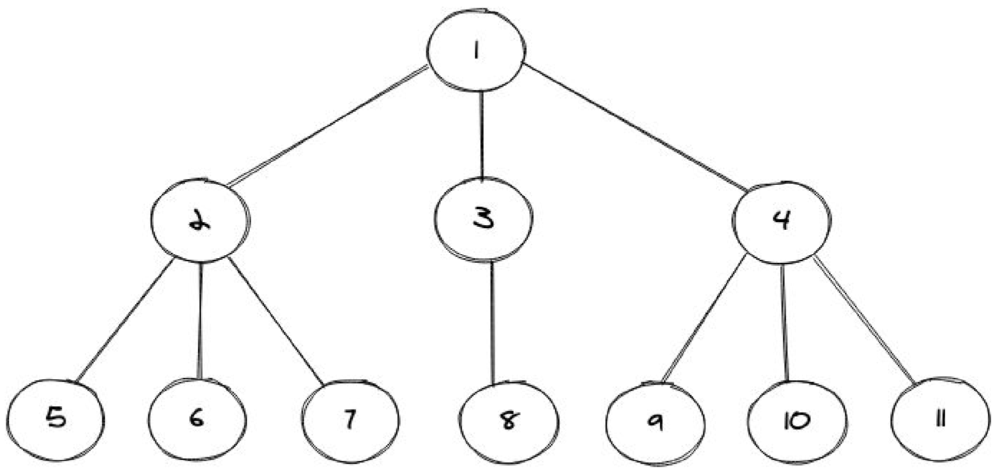
```
    50(73)
    /    \
  30(48)  60(55)
    \    \
  40(21)   70(50)
           \
           80(44)


 After deleteNode(30)
    50(73)
    /   \
  40(21)  60(55)
         \
         70(50)
          \
          80(44)


 After deleteNode(50)
    60(55)
    /   \
  40(21) 70(50)
         \
        80(44)
```

N-ary Tree

All images courtesy of https://studytonight.com

The N-ary tree is so-named because a node can have n number of children. This makes it one of the more complex types of binary search trees than standard search trees, which can only have up to two children for a node.

Here's what a N-ary tree looks like:



Here, we see that the tree has 11 nodes and some of them have three children, while others only have one. With binary trees, these children nodes are easier to store as two nodes can be assigned – left and right – to a node, making each child. With a N-ary tree, it isn't so easy. For a node's children to be stored, we need to use a different data structure – in Java, it is the LinkedList, and in C++, it is the vector.

Implementation

With a non-linear data structure, we first need to create a structure (in Java, this would be a constructor) for the data structure. As with a binary search tree, we can use the TreeNode class and create the constructors inside it with class-level variables.

Have a look at this example:

```
public static class TreeNode{
    int val;
```

```
        List<TreeNode> children = new LinkedList<>();

        TreeNode(int data){
            val = data;
        }

        TreeNode(int data,List<TreeNode> child){
            val = data;
            children = child;
        }
    }
```

Here, there is a TreeNode class. This has two constructors in it with the same name, but they are overloaded. This means that, while they share the same name, they have different parameters. There are two identifiers – val and List. Val stores a node's value while List stores the node's children nodes.

That code gives us the basic N-ary tree structure, so we have to make the tree and then use level order traversal to print it. The constructors defined in the two classes above are used to build the tree:

```
    public static void main(String[] args) {
        // creating a replica of the above N-ary Tree
        TreeNode root = new TreeNode(1);
        root.children.add(new TreeNode(2));
        root.children.add(new TreeNode(3));
        root.children.add(new TreeNode(4));
        root.children.get(0).children.add(new TreeNode(5));
        root.children.get(0).children.add(new TreeNode(6));
        root.children.get(0).children.add(new TreeNode(7));
        root.children.get(1).children.add(new TreeNode(8));
        root.children.get(2).children.add(new TreeNode(9));
                root.children.get(2).children.add(new
    TreeNode(10));
                root.children.get(2).children.add(new
    TreeNode(11));
        printNAryTree(root);
    }
```

First, the N-ary tree's root node was created. Then we assigned the root node some children, using the dot operator (.) and accessing the root node's children property. We used the add() method in the List interface to add the children to the root nodes.

Once all the children are added, we can add the children of all the level nodes. This is done by using the get()method from the List interface to access the node and add the right children to the node.

Lastly, we call the printNAryTreeMethod to print it.

Now, printing trees is not quite so easy as you would think it is. Instead of a loop through a series of items, we need to use different techniques known as algorithms. These are:

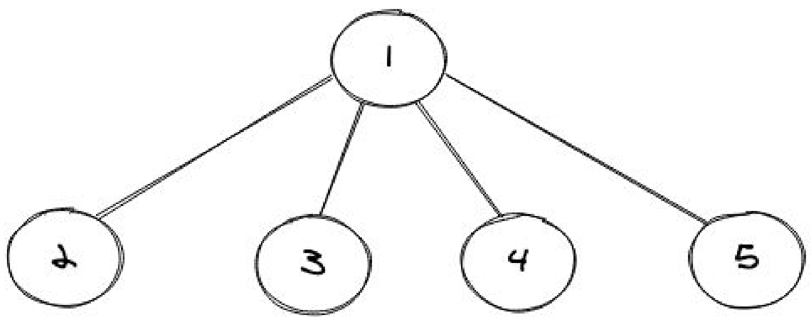
- Inorder Traversal
- Preorder Traversal
- PostOrder Traversal
- Level Order Traversal

For simplicity's sake, we'll stick with using the Level Orde traversal as it's the easiest to understand, provided you have already seen it at work on a binary search tree.

Level Order Traversal

Level Order Traversal considers that we want the root level nodes printed first before moving onto the next level and so on until we reach the final level. The nodes are stored at a particular level using the Queue data structure.

Have a look at the example tree below:



On this tree, Level Order Traversal looks like this:

```
1
2 3 4 5
```

Have a look at the following code:

```
private static void printNAryTree(TreeNode root){
        if(root == null) return;
        Queue<TreeNode> queue = new LinkedList<>();
        queue.offer(root);
        while(!queue.isEmpty()) {
           int len = queue.size();
              for(int i=0;i<len;i++) { // so that we can reach each level
              TreeNode node = queue.poll();
              System.out.print(node.val + " ");
                 for (TreeNode item : node.children) { // for-Each loop will iterate over all the children
                 queue.offer(item);
              }
           }
           System.out.println();
```

```
            }
        }
```

The whole code would look like this:

```
        import java.util.LinkedList;
        import java.util.List;
        import java.util.Queue;

        public class NAryTree {
            public static class TreeNode{
                int val;
                List<TreeNode> children = new LinkedList<>();

                TreeNode(int data){
                    val = data;
                }

                TreeNode(int data,List<TreeNode> child){
                    val = data;
                    children = child;
                }
            }

            private static void printNAryTree(TreeNode root){
                if(root == null) return;
                Queue<TreeNode> queue = new LinkedList<>();
                queue.offer(root);
                while(!queue.isEmpty()) {
```

```
                int len = queue.size();
                for(int i=0;i<len;i++) {
                    TreeNode node = queue.poll();
                    assert node != null;
                    System.out.print(node.val + " ");
                    for (TreeNode item : node.children) {
                        queue.offer(item);
                    }
                }
                System.out.println();
            }
        }

        public static void main(String[] args) {
            TreeNode root = new TreeNode(1);
            root.children.add(new TreeNode(2));
            root.children.add(new TreeNode(3));
            root.children.add(new TreeNode(4));
            root.children.get(0).children.add(new TreeNode(5));
            root.children.get(0).children.add(new TreeNode(6));
            root.children.get(0).children.add(new TreeNode(7));
            root.children.get(1).children.add(new TreeNode(8));
            root.children.get(2).children.add(new TreeNode(9));
                        root.children.get(2).children.add(new
TreeNode(10));
                        root.children.get(2).children.add(new
TreeNode(11));
            printNAryTree(root);
        }
    }
```

Output:

```
1
2 3 4
5 6 7 8 9 10 11
```

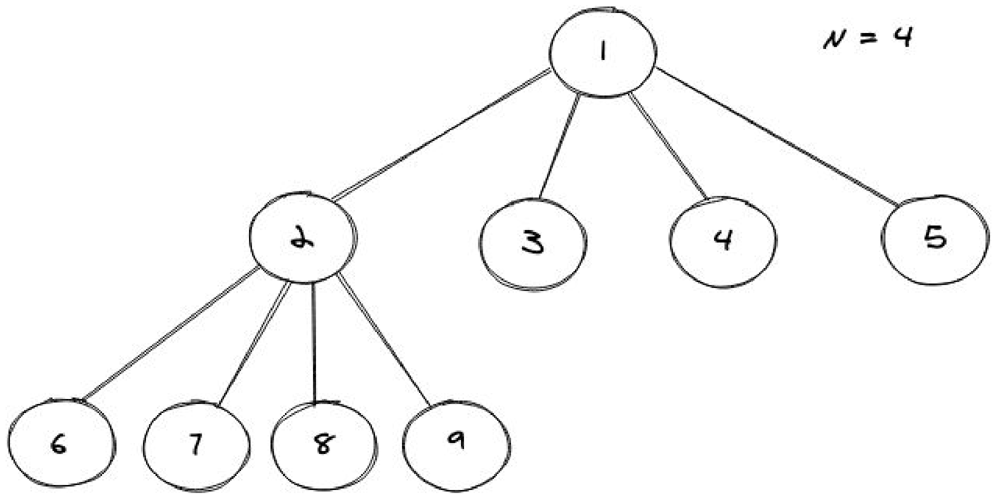
This output can be compared to the first N-ary tree we showed you, with each level node having the same values.

Types of N-ary Tree

These are the N-ary tree types you will come across:

1. **Full N-ary Tree**

This type of N-ary tree allows a node to have N or 0 children. Here's a representation:

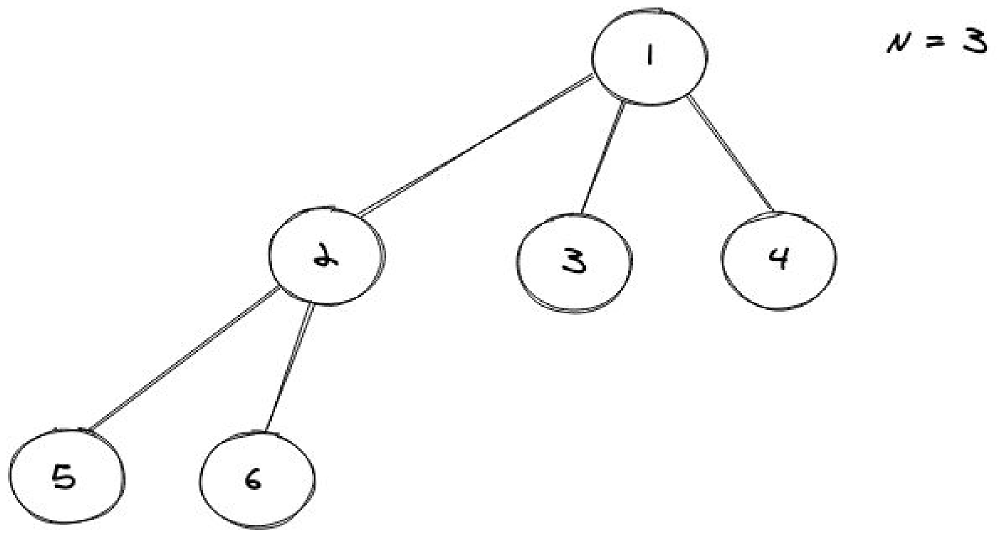


Here, you can see that the nodes satisfy the property by having 0 or 4 children.

2. **Complete N-ary Tree**

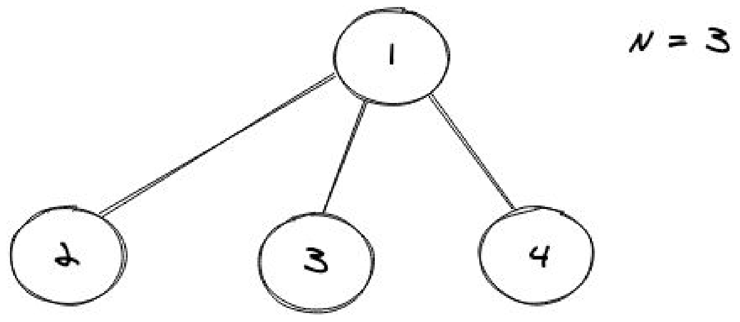
This type of N-ary tree is one where each level's nodes should have N children exactly, thus being complete. The exception is the last level nodes – if these are not complete, they should be "as left as possible."

Here’s a representation:



3. **Perfect N-ary Tree**

Perfect N-ary trees are full trees with the leaf node levels all being the same. Here’s a representation:



Part 3: Disjoint Sets

Disjoint Set Data Structures

In computer science, a disjoint-set data structure is also known as a merge-find set or a union-find data structure. These data structures store collections of disjoint sets, which are non-overlapping sets. In each disjoint subset, it will also store a partition of a set. These data structures provide the operations needed to add new sets, merge sets, or find representative set members. The latter operation gives us an efficient way to see if two elements share a set or are in different ones.

We can implement a disjoint-set data structure in several ways, but the most common method is with a disjoint-set forest. These are special forests that allow unions and finds to be performed in near-constant amortized time. A sequence of m addition, find, or union operation performed with n nodes on a disjoint-set data structure needs a total time of O(mα(n). In this, α(n) is the inverse Ackermann function, one of the slowest growing.

However, keep in mind that this performance is not guaranteed on a per-operation basis by the disjoint-set forest. Some find and union operations take much longer than the constant α(n) time, but every individual operation will make the forest adjust itself to speed up subsequent operations. Disjoint-set forests are efficient and asymptotically optimal.

These data structures play an important role in Kruskal's algorithm, which finds a graph's minimum spanning tree. These trees have an importance that means the disjoint-set structure is used in many different algorithms.

Representation

In a disjoint-set forest, every node will have two things – a pointer and some supplemental information. This is usually a rank or a size but never both. The pointers are used to create parent pointer trees. In these trees, where a node isn't a tree root, it will point to its parent node. To ensure parent nodes are easy to distinguish from other nodes, invalid values are supplied to the parent pointers, such as a sentinel value or a circular reference.

Each tree represents a set in the forest, and the tree nodes are the members of those sets. A root node has set representatives – two nodes can share a set provided the tree roots with those nodes are equal.

The nodes can be stored in the forest in any way that suits the application. However, one of the commonest techniques is storing the nodes in arrays. The array index will indicate the parents in this case. Each entry in the array needs Θ(log n) bits of storage for their respective parent pointers, while the remainder of the entry needs the same or less storage. This means storing the forest takes Θ(n log n) bits. If fixed-size nodes are used in the implementation, resulting in the forest being restricted to a maximum storage size, linear in n is the right storage type.

Operations

Each disjoint-set data structure provides support for three operations:

- Creating new sets with new elements
- Locating the set's representative with a specified element
- Merging two sets

Let's look at these in turn.

1. **Creating New Sets**

New elements are added using the MakeSet operation. The new element is then put into a new set with just a single element, and this set is added to the disjoint-set data structure. If we were to view the structure as a partition of a set, the MakeSet operation would add the new element and make the set bigger, extending the partition by adding the element to a new subset with a single element – the new one.

In the disjoint-set forest, the operation initializes the parent pointer and the size or rank of the node. If a node pointing to itself represents the root, we could use the pseudocode below to add a new element:

```
function MakeSet(x) is
    if x is not in the forest already, then
        x.parent := x
        x.size := 1     // if nodes store size
        x.rank := 0     // if nodes store rank
    end if
end function
```

The time complexity for the operation is constant and, if you initialize a forest with n nodes, it will need O(n) time.

In reality, you must precede the MakeSet operation using another operation to allocate memory, holding x. The random-set forest's asymptotic performance will not change so long as the memory allocation operation is in amortized constant time, the same as when you implement a dynamic array.

2. **Locate a Representative**

When you use the Find operation, it must follow a specific path – from the given query node x, it must follow a series of parent pointers until it gets to a root element. This element is representative of the set x belongs in and could potentially be x. The Find operation will return the root element.

The Find operation is a useful way to improve the forest. The majority of the time involves following the parent pointers so, if you flatten the tree, it will take less time to do the Find operation. When you execute Find, the quickest way to reach the root is to follow the parent pointers in order. However, these particular parent pointers can be updated so they point nearer to the root. Each element you visit as you make your way to the root is in the same set, so the sets that the forest stores are not changed in any way. However, it does speed up successive Find operations for the nodes between the root and query point and all their descendants. The mortised performance guarantee relies heavily on this updating.

Several algorithms for the Find operation satisfy the asymptotic optimal time complexity. Path compression is a family of algorithms that make all the nodes between the root and query point to the root. We can implement path regression with the following recursion:

```
function Find(x) is
    if x.parent ≠ x then
        x.parent := Find(x.parent)
        return x.parent
    else
        return x
    end if
end function
```

You will see that two passes are made in the implementation – up the tree and back down it. Sufficient scratch memory is required for the path between the query and root node to be stored. In the code above, the call stack is used to represent the path implicitly. If both passes are performed in the same direction, we can decrease this to constant memory. This is implemented by traversing the query to the root node path twice, the first time to find the root and the second to update the pointers:

```
function Find(x) is
    root := x
    while root.parent ≠ root do
        root := root.parent
    end while

    while x.parent ≠ root do
        parent := x.parent
        x.parent := root
        x := parent
    end while


    return root
end function
```

One-pass Find algorithms were also developed to retain the worst-case complexity but are inherently more efficient. These are known as path halving and path splitting, and both will update the parent pointers for those nodes found on the path from the query to the root node. Path splitting will replace all of that path's parent pointers with a pointer to the grandparent for the specific node:

```
function Find(x) is
    while x.parent ≠ x do
        (x, x.parent) := (x.parent, x.parent.parent)
    end while
    return x
end function
```

While path halving works in much the same way, it only replaces every alternative parent pointer:

```
function Find(x) is
    while x.parent ≠ x do
        x.parent := x.parent.parent
        x := x.parent
    end while
    return x
end function
```

Merging Two Sets

The Union(x, y) operation replaces two sets – one with x and one with y – with a union. This operation used Find to work out the roots in the trees that contain x and y. If they are the same roots, that's all there is to do. However, if the roots are not the same, the next step is to merge the trees. We do this in one of two ways – setting the x root's pointer parent to y's or vice versa.

However, your choice of which node will become the parent may have consequences for future tree operation complexity. Chosen carelessly, a tree can become too tall. For example, let's assume that the tree with x is always made a subtree of the one containing y by the Union operation. We start with a forest initialized with its elements and then execute the following:

Union(1, 2), Union(2, 3), …, Union(n -1, n)

The result is a forest with just one tree, and that tree has a root of n. The path between 1 and n goes through all the nodes in the tree. The time taken to run Find(1) would be O(n).

If your implementation is efficient, Union by Rank or Union by Size are used to control the tree height. Both require that nodes store both a parent pointer and additional information, and this information determines the root that will become the parent. Both of these ensure that we do not end up with an excessively deep tree.

With Union by Size, the node's size is stored in the node. This is nothing more than a number representing how many descendants there are, including the node. When the trees with the x and y roots are merged, the parent node is the one with the most descendants. If both nodes have an identical number of descendants, either can be the parent, but, in both cases, the parent node is set to the number representing the total descendants:

```
function Union(x, y) is
    // Replace nodes by roots
    x := Find(x)
    y := Find(y)
```

```
        if x = y then
            return  // x and y are already in the same set
        end if

        // If needed, rename the variables to make sure that
        // x has at least as many descendants as y
        if x.size < y.size then
            (x, y) := (y, x)
        end if

        // Make x the new root
        y.parent := x
        // Update the size of x
        x.size := x.size + y.size
    end function
```

The number of bits needed to store node size is obviously the number needed to store n. This provides the required storage for the forest with a constant factor.

With Union by Rank, the node's rank is stored by the node and is the height's upper bound. The first step in merging the trees with the x and y roots is to compare their respective ranks. If they are not the same, the larger one will be the parent, and the x and y ranks will remain the same. However, if they are the same, either can be the parent, but the parent rank will increase by 1. While there is a close relationship between a node's rank and height, it is always more efficient to store ranks rather than height. During a Find operation, a node's height may change. Because of that, storing a rank means not having to do the extra work needed to keep the height correct.

The pseudocode for Union by Rank is:

```
function Union(x, y) is
    // Replace nodes by roots
    x := Find(x)
    y := Find(y)

    if x = y then
        return  // x and y are in the same set
    end if

    // If needed, rename the variables to ensure that
    // x has rank at least as large as that of y
    if x.rank < y.rank then
        (x, y) := (y, x)
    end if

    // Make x the new root
    y.parent := x
    // If needed, increment the rank of x
    if x.rank = y.rank then
        x.rank := x.rank + 1
    end if
end function
```

All nodes have a rank of (logN) or lower, and, as a result, we can store the rank in O(log log n) bits. This means it is an asymptotically negligible part of the overall size of the forest.

From these implementations, you can clearly see that the node's rank and size are not important unless the node is a tree's root. Once the becomes a child, you will never access its size and rank again.

Time Complexity

Disjoint-set forest implementations where the parent pointers are not updated by the Find operation and where the tree heights are not controlled by Union may contain trees with O(n) height. In this situation, O(n) is required

Where an implementation uses only path compression, the worst-case running time of a sequence containing n MakeSet operations and then no more than n – 1 Union operations and f Find operations is $\theta(n + f . (1 + \log_{2+f/n} n))$.

When you use Union by Rank, but the parent pointers are not updated during the Find operation, the running time of $\theta(m \log n)$ for any type of m operations up to n (MakeSet operations) to $\theta(m\alpha(n))$. This ensures each operation has an amortized running time of $\theta(\alpha(n))$, asymptotically optimal, ensuring every disjoint-set structure uses an amortized time per operation of $\Omega(\alpha(n))$.

In this, the inverse Ackerman function is $\alpha(n)$, an extraordinarily slow grower, so the factor for any n written in the physical universe is 4 or lower, giving all disjoint-set operations an amortized constant time.

Part 4: Advanced Heaps and Priority Queues

Binary Heap or Binary Search Tree for Priority Queues?

Binary heaps are always the preferred option for priority queues over binary search trees. For a priority queue to be efficient, the following operations are required:

1. Get the minimum or maximum to determine the top priority element
2. Insert an element
3. Eliminate the top priority element
4. Decrease the key

Binary heaps support these operations with the following respective time complexities:

1. O(1)
2. O(Logn)
3. O(Logn)
4. O(Logn)

The red-black, AVL, and other self-balancing binary search trees also support these operations, sharing the same time complexities.

Let's break down those operations:

1. O(1) is not naturally the time complexity for finding the minimum and maximum. However, by retaining an additional pointer to the min or max and updating it with deletion or insertion as needed, it can be implemented within that time complexity. When we use deletion, the update is done by finding the inorder successor or predecessor.
2. O(Logn) is the natural time complexity for inserting elements.
3. O(Logn) is also natural for removing the minimum and maximum.
4. We can decrease the key in O(Logn) with a sequence of a deletion and an insertion.

But how does this answer our question – why are binary heaps preferred for the priority queues?

- Because arrays are used to implement binary heaps, the locality of reference is always better, and we get more cache-friendly operations.
- Although the operations share the same time complexities, binary search tree constants are always higher.
- A binary heap can be built in O(n) time, while a self-balancing binary search tree needs O(nLogn) time.
- Binary heaps don’t need additional space for the pointers like binary search trees do.
- Binary heaps are much easier to implement than binary search trees.
- Binary heaps come in several variations, such as the Fibonacci Heap that supports insert and decrease-key in θ(1) time.

But are binary heaps always better than a binary search tree?

Sure, binary heaps are better for priority queues, but binary search trees have a much bigger list of advantages in other ways:

- It takes O(Logn) to search for an element in a self-balancing binary search tree, while the same in a binary heap takes O(n) time.
- All the elements in a binary search tree can be printed in sorted order in O(n) time, while the binary heap takes (O(nLogn) time.
- Adding an additional field to the binary search tree makes the kth smallest or largest element take O(Logn) time.

Now we’ve had a quick look at why binary heaps are better for priority queues than the binary search tree, it’s time to dig into some of the different types.

Binomial Heap

Binary heaps have one primary application – to implement the priority queues. The binomial heap is an extension of the binary heap, providing much faster merge or union operations in addition to all the other operations the binary heap offers.

What Is a Binomial Heap?

Let's say you have a binomial tree with an order of 0 – it would have a single node. Constructing a binomial tree with an order k requires two binomial trees with an order k-1 and setting one of them as the left child. This tree with an order of k has these properties:

- It has 2^k nodes exactly
- Its depth is k
- At depth i for i – 0, 1, …, k, it has ^kc_i nodes exactly
- The root of the tree has a degree of k, and the root's children are binomial trees with the following order:

k-1, k-2,.. 0 left to right.

k = 0 (Single Node)

```
o
```

k = 1 (2 nodes)

[Take two k = 0 order Binomial Trees, and

set one of them as a child of the other]

```
 o
/
o
```

k = 2 (4 nodes)

[Take two k = 1 order Binomial Trees, and set one of them as a child of the other]

```
      o
    /   \
   o     o
 /
o
```

k = 3 (8 nodes)

[Take two k = 2 order Binomial Trees, and

set one of them as a child of the other]

```
       o
    /  | \
   o    o o
 / \  |
o   o o
      \
       o
```

The Binomial Heap

Binomial heaps are nothing more than a series of binomial trees. Each tree will follow the Min Heap property, and there can only be one tree t most of any particular degree. Here are some examples:

```
12——10————20
       /  \           /  | \
      15   50          70  50  40
      |              / |   |
      30              80  85  65
                   |
                    100
```

This binomial heap contains 13 nodes and is a series of three trees with orders from left to right of 0, 2, and 3.

```
   10————20
  / \          / |\
15   50         70  50  40
|            /|   |
30           80  85  65
             |
            100
```

This binomial heap contains 12 nodes and is a series of two trees with orders from left to right of 2 and 3.

Representing Numbers and Binomial Heaps in Binary

Let's say you have a binomial heap with n nodes. It will contain binomial trees that equal the number of bits in n's binary representation. For example, we'll say that n is 13 and the binary representation of n, which is 00001101, has three set bits. That means there are three binomial trees. The degrees of these trees can also be related to the positions of the set bits, and, using that relationship, we conclude that a binomial heap with n nodes has O(Logn) binomial trees.

Binomial Heap Operations

The binomial heap's primary function is union(), and all other operations also use this operation. We use the union() operation to combine two heaps into a single one. We'll discuss union later; first, the other operations in the binomial heap:

1. **insert(H, k)** – this operation inserts key 'k' into the binomial heap, 'H,' creating a binomial heap that has a single key. Next, union is called on H and the newly created binomial heap.

2. **getMin(H)** – the easiest way of obtaining getMin() is by traversing the root list of the trees and returning the smallest or minimum key. Implementing this requires O(Logn) time, but we can optimize this to O(1) by ensuring a pointer to the minimum key root is maintained.
3. **extractMin(H)** – extractMin() also uses the union() operation. First, getMin() is called to obtain the minimum key tree. Then the node is removed, and a new binomial heap is created by all the subtrees from the removed node being connected. Lastly, union() is called on H and the new heap. This entire operation needs O(Logn) time.
4. **delete(H)** – similar to the binary heap, the delete operation takes two steps – the key is reduced to minis infinite, and then extractMin() is called.
5. **decreaseKey(H)** – this is also like the binary heap. The decrease key is compared with its parent. Should the parent key be greater, the keys are swapped and recur for the parent key. The operation stops when it reaches a node where the parent ey is smaller, or it reaches the root node. decreaseKey() has a time complexity of O(Logn).

Union Operation

Let's say we have a pair of binomial heaps, H2 and H2. The union() operation will create one binomial heap using the following steps:

1. Step one is merging the heaps by a non-decreasing order of the heap degrees.
2. Once this is done, it's time to ensure there is no more than one binomial tree of any order. This requires binomial trees sharing the same order to be combined. The list of the merged roots is traversed, and we track the three pointers called prev, x, and next-x. There are four cases where the list of roots is traversed:
   i. Where the x and next-x orders are not the same, we do nothing more than move in

ii. Where the next-next-x order is the same, move on

iii. If x’s key is equal to or less than the next-x key, next-x is made into x’s child – the two are linked together to achieve this.

iv. If x’s key is greater, x is made next’s child.

Binomial Heap Implementation

The following implementation in C++ includes the insert(), getMin(), andextractMin() operations:

```
// C++ program to implement some operations

// on Binomial Heap

#include<bits/stdc++.h>

using namespace std;


// A Binomial Tree node.

struct Node

{

    int data, degree;

    Node *child, *sibling, *parent;

};


Node* newNode(int key)

{

    Node *temp = new Node;

    temp->data = key;

    temp->degree = 0;

    temp->child = temp->parent = temp->sibling =
NULL;

    return temp;
```

```
}

// This function is used to merge two Binomial Trees.
Node* mergeBinomialTrees(Node *b1, Node *b2)
{
    // b1 must be smaller
    if (b1->data > b2->data)
        swap(b1, b2);

    // We set a larger valued tree
    // as a child of a tree with a smaller value
    b2->parent = b1;
    b2->sibling = b1->child;
    b1->child = b2;
    b1->degree++;

    return b1;
}

// This function is used to perform the union operation
on
// two binomial heaps i.e. l1 & l2
list<Node*> unionBionomialHeap(list<Node*> l1,
                              list<Node*> l2)
{
    // _new to another binomial heap with a
    // new heap after merging l1 & l2
```

```
list<Node*> _new;
list<Node*>::iterator it = l1.begin();
list<Node*>::iterator ot = l2.begin();
while (it!=l1.end() && ot!=l2.end())
{
    // if D(l1) <= D(l2)
    if((*it)->degree <= (*ot)->degree)
    {
        _new.push_back(*it);
        it++;
    }
    // if D(l1) > D(l2)
    else
    {
        _new.push_back(*ot);
        ot++;
    }
}

// if some elements remain in l1
// binomial heap
while (it != l1.end())
{
    _new.push_back(*it);
    it++;
}
```

```
    // if some elements remain in l2
    // binomial heap
    while (ot!=l2.end())
    {
        _new.push_back(*ot);
        ot++;
    }
    return _new;
}

// the adjust function will rearrange the heap so that
// the heap is in increasing order of degree and
// no two binomial trees have the same degree in this
heap
list<Node*> adjust(list<Node*> _heap)
{
    if (_heap.size() <= 1)
        return _heap;
    list<Node*> new_heap;
    list<Node*>::iterator it1,it2,it3;
    it1 = it2 = it3 = _heap.begin();

    if (_heap.size() == 2)
    {
        it2 = it1;
        it2++;
        it3 = _heap.end();
    }
    else
    {
        it2++;
        it3=it2;
        it3++;
    }
    while (it1 != _heap.end())
    {
        // if only a single element remains to be processed
        if (it2 == _heap.end())
            it1++;

        // If D(it1) < D(it2) i.e. merging of Binomial
        // Tree pointed by it1 & it2 is not possible
        // then move next in heap
        else if ((*it1)->degree < (*it2)->degree)
        {
            it1++;
            it2++;
            if(it3!=_heap.end())
                it3++;
        }

        // if D(it1),D(it2) & D(it3) are same i.e.
        // degree of three consecutive Binomial Tree are
same
```

```
        // in heap
        else if (it3!=_heap.end() &&
                (*it1)->degree == (*it2)->degree &&
                (*it1)->degree == (*it3)->degree)
        {
            it1++;
            it2++;
            it3++;
        }

        // if the degrees of two Binomial Trees are the
same in the heap
        else if ((*it1)->degree == (*it2)->degree)
        {
            Node *temp;
            *it1 = mergeBinomialTrees(*it1,*it2);
            it2 = _heap.erase(it2);
            if(it3 != _heap.end())
                it3++;
        }
    }
    return _heap;
}

// inserting a Binomial Tree into binomial heap
list<Node*> insertATreeInHeap(list<Node*> _heap,
                              Node *tree)
```

```
{
    // creating a new heap i.e temp
    list<Node*> temp;

    // inserting Binomial Tree into heap
    temp.push_back(tree);

    // perform union operation to finally insert
    // Binomial Tree in original heap
    temp = unionBionomialHeap(_heap,temp);

    return adjust(temp);
}

// removing the minimum key element from the
binomial heap
// this function takes the Binomial Tree as an input and
returns
// a binomial heap after
// removing the head of that tree, i.e., minimum element
list<Node*> removeMinFromTreeReturnBHeap(Node
*tree)
{
    list<Node*> heap;
    Node *temp = tree->child;
    Node *lo;

    // creating a binomial heap from Binomial Tree
    while (temp)
    {
        lo = temp;
        temp = temp->sibling;
        lo->sibling = NULL;
        heap.push_front(lo);
    }
    return heap;
}

// inserting a key into the binomial heap
list<Node*> insert(list<Node*> _head, int key)
{
    Node *temp = newNode(key);
    return insertATreeInHeap(_head,temp);
}

// return a pointer of minimum value Node
// present in the binomial heap
Node* getMin(list<Node*> _heap)
{
    list<Node*>::iterator it = _heap.begin();
    Node *temp = *it;
    while (it != _heap.end())
    {
        if ((*it)->data < temp->data)
            temp = *it;
        it++;
    }
    return temp;
}

list<Node*> extractMin(list<Node*> _heap)
{
    list<Node*> new_heap,lo;
    Node *temp;

    // temp contains the pointer of minimum value
    // element in heap
    temp = getMin(_heap);
    list<Node*>::iterator it;
    it = _heap.begin();
    while (it != _heap.end())
    {
        if (*it != temp)
        {
            // inserting all Binomial Tree into new
            // binomial heap except the Binomial Tree
            // contains minimum element
            new_heap.push_back(*it);
        }
        it++;
    }
    lo = removeMinFromTreeReturnBHeap(temp);
    new_heap = unionBionomialHeap(new_heap,lo);
    new_heap = adjust(new_heap);
    return new_heap;
}

// print function for Binomial Tree
void printTree(Node *h)
{
    while (h)
    {
        cout << h->data << ” “;
        printTree(h->child);
        h = h->sibling;
    }
}

// print function for binomial heap
void printHeap(list<Node*> _heap)
{
    list<Node*> ::iterator it;
    it = _heap.begin();
    while (it != _heap.end())
    {
        printTree(*it);
        it++;
    }
}
```

```
// Driver program to test above functions
int main()
{
    int ch,key;
    list<Node*> _heap;

    // Insert data in the heap
    _heap = insert(_heap,10);
    _heap = insert(_heap,20);
    _heap = insert(_heap,30);

    cout << “Heap elements after insertion:\n”;
    printHeap(_heap);

    Node *temp = getMin(_heap);
    cout << “\nMinimum element of heap “
        << temp->data << “\n”;

    // Delete minimum element of heap
    _heap = extractMin(_heap);
    cout << “Heap after deletion of minimum element\n”;
    printHeap(_heap);

    return 0;
}
```

Output:

```
The heap is:
50 10 30 40 20
After deleing 10, the heap is:
20 30 40 50
```

Fibonacci Heap

As you know, the primary use for a heap is to implement a priority queue, and the Fibonacci heap beats the binomial and binary heaps when it comes to time complexity. Below, you can see the Fibonacci heap's amortized time complexities:

1. **Find Min -** Θ(1), which is the same as the binomial and binary heaps
2. **Delete Min -** O(Logn), which is Θ(Logn) in the binomial and binary heaps
3. **Insert -** Θ(1), which is Θ(Logn) in the binary heap and Θ(1) in the binomial heap
4. **Decrease-Key -** Θ(1), which is Θ(Logn) in the binomial and binary heaps
5. **Merge -** Θ(1), which is Θ(m Logn) or Θ(m+n) in the binary heap and Θ(Logn) in the binomial heap

The Fibonacci heap is like the binomial heap in that it is a series of trees that have the max-heap or min-heap property. In Fibonacci, the trees can be of any shape and can even all be single nodes, as opposed to the binomial heap where all the trees must be binomial.

A pointer to the minimum value (tree root) is also maintained in the Fibonacci heap, and a doubly-linked list is used to connect all the trees. That way, a single pointer to min can be used to access all of them.

The primary idea is to use a "lazy" way of executing operations. For example, the merge operation does nothing more than link two heaps, while the insert operation is no more complicated than adding a new tree with just a single node. The most complicated is the extract minimum operation, as it is used to consolidate trees. This results in the delete operation also being somewhat complicated as it must first decrease the key to minimum infinite before calling the extract minimum operation.

Interesting Points

1. The Decrease-Key operation has a reduced time complexity, which is important to the Prim and Djikstra algorithms. If you use a binary heap, these algorithms have a time complexity of O(VLogV + ELogV), but if you use the Fibonacci heap, that improves to time complexity of O(VLogV + E).
2. While the time complexity looks promising for the Fibonacci heap, it has proved to be somewhat slow in real-world practice because of high hidden constants.
3. Fibonacci heaps gain their name primarily from the fact that the running time analysis uses Fibonacci numbers. Added to that is the fact that all the nodes in a Fibonacci heap have a maximum O(Logn) degree. And, when a subtree has its root in a node of degree k, its size is a minimum of F_{k+2}, and the kth Fibonacci number is F_k.

Insertion and Union

Here, we are going to discuss two of the Fibonacci heap operations, starting with insertion.

Insertion

We use the algorithm below to insert a new node into a Fibonacci heap:

1. A new node, x, is created
2. Heap H is checked to see if it is empty or not
3. If it is, x is set as the single node in the root list, and the H(min) pointer is set to x
4. If the heap isn't empty, x is inserted into the root list, and H(min) is updated.

Union

We can do a union of H1 and H2, both Fibonacci heaps, using the following algorithm:

1. The root lists of H1 and H2 are joined into one Fibonacci heap H
2. If H1(min) < H2(min) then H(min) = H2(min)

3. If not, H(min) = H1(min)

The program below uses C++ to build a Fibonacci heap and insert into it:

```
// C++ program to demonstrate building
// and a Fibonacci heap and inserting into it
#include <cstdlib>
#include <iostream>
#include <malloc.h>
using namespace std;

struct node {
    node* parent;
    node* child;
    node* left;
    node* right;
    int key;
};

// Create min pointer as “mini”
struct node* mini = NULL;

// Declare an integer for the number of nodes in the heap
int no_of_nodes = 0;

// Function to insert a node in the heap
void insertion(int val)
{
    struct node* new_node = (struct
node*)malloc(sizeof(struct node));
    new_node->key = val;
    new_node->parent = NULL;
    new_node->child = NULL;
    new_node->left = new_node;
    new_node->right = new_node;
    if (mini != NULL) {
        (mini->left)->right = new_node;
        new_node->right = mini;
        new_node->left = mini->left;
        mini->left = new_node;
        if (new_node->key < mini->key)
            mini = new_node;
    }
    else {
        mini = new_node;
    }
}

// Function to display the heap
void display(struct node* mini)
{
    node* ptr = mini;
    if (ptr == NULL)
        cout << “The Heap is Empty” << endl;
    else {
        cout << “The root nodes of Heap are: ” << endl;
        do {
            cout << ptr->key;
            ptr = ptr->right;
            if (ptr != mini) {
                cout << “—>”;
            }
        } while (ptr != mini && ptr->right != NULL);
        cout << endl
             << “The heap has ” << no_of_nodes << ” nodes”
<< endl;
    }
}
// Function to find the min node in the heap
void find_min(struct node* mini)
{
    cout << “min of heap is: ” << mini->key << endl;
}



// Driver code
int main()
{


    no_of_nodes = 7;
    insertion(4);
    insertion(3);
    insertion(7);
    insertion(5);
    insertion(2);
    insertion(1);
    insertion(10);

    display(mini);

    find_min(mini);

    return 0;
}
```

Output:

```
The root nodes of Heap are:
1—>2—>3—>4—>7—>5—>10
The heap has 7 nodes
Min of heap is: 1
```

Fibonacci Heap – Deletion, Extract min and Decrease key

We've just looked at insertion and union, both operations being critical to understanding the next three operations. We'll start with Extract_min().

Extract_min()

In this operation, a function is created to delete the minimum node. Then the min pointer is set to the minimum value in the rest of the heap. The algorithm below is used:

1. The min node is deleted
2. The head is set to the next min node. Then, all the trees from the deleted node are added to the root list
3. An array is created, containing degree pointers with the same size as the deleted node
4. The degree pointer is set to the current node
5. Then we move to the next node. If the degrees are the same, the union operation is used to join the trees. If they are different, the degree pointer is set to the next node.
6. Steps four and five are repeated until the heap is finished.

Decrease_key()

If we want the value of any heap element decreased, the following algorithm is followed:

1. Node x's value is decreased to the new specified value
2. Case 1 – should this result in the min-heap property not being violated, the min pointer is updated if needed
3. Case 2 – should this result in the min-heap property being violated and x's parent is not marked, we need to do three things:
    a. The link between x and the parent is cut off
    b. X's parent is marked
    c. The tree rooted at x is added to the root list and, if needed, the min pointer is updated

4. Case 3 – if there is a violation of the min-heap property and x's parent is already marked:
    a. The link is cut off between x and p[x] – the parent
    b. x is added to the root list, and the min pointer is updated if needed
    c. The link is cut off between p[x] and p[p[x]]
    d. p[x] is added to the root list, and the min pointer is updated if needed
    e. p[p[x]] is marked if it is unmarked
    f. Otherwise, p[p[x]] is cut off, and steps 4b to 4e are repeated, using p[p[x] as x.

Deletion()

The algorithm below is followed to delete an element from the heap:

1. The value of the node (x) you want to delete is decreased to a minimum using the Decrease_key() function
2. The heap with x is then heapified using the min-heap property, putting x in the root list
3. Extract_min() is applied to the heap

Below is a program in C++ to demonstrate these operations:

```
// C++ program to demonstrate the Extract min, Deletion()

// and Decrease key() operations

#include <cmath>

#include <cstdlib>

#include <iostream>

#include <malloc.h>

using namespace std;


// Create a structure to represent a node in the heap
```

```
struct node {
    node* parent; // Parent pointer
   node* child; // Child pointer
    node* left; // Pointer to the node on the left
    node* right; // Pointer to the node on the right
    int key; // Value of the node
    int degree; // Degree of the node
    char mark; // Black or white mark of the node
    char c; // Flag for assisting in the Find node function
};

// Creating min pointer as “mini”
struct node* mini = NULL;

// Declare an integer for the number of nodes in the heap
int no_of_nodes = 0;

// Function to insert a node in heap
void insertion(int val)
{
    struct node* new_node = new node();
    new_node->key = val;
    new_node->degree = 0;
    new_node->mark = ‘W’;
    new_node->c = ‘N’;
    new_node->parent = NULL;
    new_node->child = NULL;
    new_node->left = new_node;
    new_node->right = new_node;
    if (mini != NULL) {
        (mini->left)->right = new_node;
        new_node->right = mini;
        new_node->left = mini->left;
        mini->left = new_node;
        if (new_node->key < mini->key)
            mini = new_node;
  }
    else {
        mini = new_node;
    }
    no_of_nodes++;
}
// Link the heap nodes in the parent child relationship
void Fibonnaci_link(struct node* ptr2, struct node* ptr1)
{
    (ptr2->left)->right = ptr2->right;
    (ptr2->right)->left = ptr2->left;
    if (ptr1->right == ptr1)
        mini = ptr1;
    ptr2->left = ptr2;
    ptr2->right = ptr2;
    ptr2->parent = ptr1;
    if (ptr1->child == NULL)
        ptr1->child = ptr2;
    ptr2->right = ptr1->child;
    ptr2->left = (ptr1->child)->left;
    ((ptr1->child)->left)->right = ptr2;
    (ptr1->child)->left = ptr2;
    if (ptr2->key < (ptr1->child)->key)
        ptr1->child = ptr2;
    ptr1->degree++;
}
// Consolidate the heap
void Consolidate()
{
    int temp1;
    float temp2 = (log(no_of_nodes)) / (log(2));
    int temp3 = temp2;
    struct node* arr[temp3+1];
    for (int i = 0; i <= temp3; i++)
        arr[i] = NULL;
    node* ptr1 = mini;
    node* ptr2;
    node* ptr3;
    node* ptr4 = ptr1;
    do {
        ptr4 = ptr4->right;
        temp1 = ptr1->degree;
        while (arr[temp1] != NULL) {
            ptr2 = arr[temp1];
            if (ptr1->key > ptr2->key) {
```

```
            ptr3 = ptr1;
            ptr1 = ptr2;
            ptr2 = ptr3;
        }
        if (ptr2 == mini)
            mini = ptr1;
        Fibonnaci_link(ptr2, ptr1);
        if (ptr1->right == ptr1)
            mini = ptr1;
        arr[temp1] = NULL;
        temp1++;
    }
    arr[temp1] = ptr1;
    ptr1 = ptr1->right;
} while (ptr1 != mini);
mini = NULL;
for (int j = 0; j <= temp3; j++) {
    if (arr[j] != NULL) {
        arr[j]->left = arr[j];
        arr[j]->right = arr[j];
        if (mini != NULL) {
            (mini->left)->right = arr[j];
            arr[j]->right = mini;
            arr[j]->left = mini->left;
            mini->left = arr[j];
            if (arr[j]->key < mini->key)
                mini = arr[j];
            }
            else {
                mini = arr[j];
            }
            if (mini == NULL)
                mini = arr[j];
            else if (arr[j]->key < mini->key)
                mini = arr[j];
        }
    }
}

// Function to extract thr minimum node in the heap
void Extract_min()
{
    if (mini == NULL)
        cout << "The heap is empty" << endl;
    else {
        node* temp = mini;
        node* pntr;
        pntr = temp;
        node* x = NULL;
        if (temp->child != NULL) {

            x = temp->child;
            do {
                pntr = x->right;
                (mini->left)->right = x;
                x->right = mini;
                x->left = mini->left;
                mini->left = x;
                if (x->key < mini->key)
                    mini = x;
                x->parent = NULL;
                x = pntr;
            } while (pntr != temp->child);
        }
        (temp->left)->right = temp->right;
        (temp->right)->left = temp->left;
        mini = temp->right;
            if (temp == temp->right && temp->child == 
NULL)
            mini = NULL;
        else {
            mini = temp->right;
            Consolidate();
        }
        no_of_nodes—;
    }
}

// Cut a node in the heap to be placed in the root list
void Cut(struct node* found, struct node* temp)
{
    if (found == found->right)
        temp->child = NULL;

    (found->left)->right = found->right;
    (found->right)->left = found->left;
    if (found == temp->child)
        temp->child = found->right;

    temp->degree = temp->degree - 1;
    found->right = found;
    found->left = found;
    (mini->left)->right = found;
    found->right = mini;
    found->left = mini->left;
    mini->left = found;
    found->parent = NULL;
    found->mark = ‘B’;
}

// Recursive cascade cutting function
void Cascase_cut(struct node* temp)
{
    node* ptr5 = temp->parent;
    if (ptr5 != NULL) {
        if (temp->mark == ‘W’) {
            temp->mark = ‘B’;
        }
        else {
            Cut(temp, ptr5);
            Cascase_cut(ptr5);
        }
    }
}

// Function to decrease the value of a node in the heap
void Decrease_key(struct node* found, int val)
{
    if (mini == NULL)
        cout << “The Heap is Empty” << endl;

    if (found == NULL)
        cout << “Node not found in the Heap” << endl;

    found->key = val;

    struct node* temp = found->parent;
    if (temp != NULL && found->key < temp->key) {
        Cut(found, temp);
        Cascase_cut(temp);
    }
    if (found->key < mini->key)
        mini = found;
}
```

```
// Function to find the given node
void Find(struct node* mini, int old_val, int val)
{
    struct node* found = NULL;
    node* temp5 = mini;
    temp5->c = ‘Y’;
    node* found_ptr = NULL;
    if (temp5->key == old_val) {
        found_ptr = temp5;
        temp5->c = ‘N’;
        found = found_ptr;
        Decrease_key(found, val);
    }
    if (found_ptr == NULL) {
        if (temp5->child != NULL)
            Find(temp5->child, old_val, val);
        if ((temp5->right)->c != ‘Y’)
            Find(temp5->right, old_val, val);
    }
    temp5->c = ‘N’;
    found = found_ptr;
}

// Delete a node from the heap
void Deletion(int val)
{
    if (mini == NULL)
```

```
        cout << “The heap is empty” << endl;
    else {

        // Decrease the value of the node to 0
        Find(mini, val, 0);

        // Call the Extract_min function to
        // delete minimum value node, which is 0
        Extract_min();
        cout << “Key Deleted” << endl;
    }
}

// Function to display the heap
void display()
{
    node* ptr = mini;
    if (ptr == NULL)
        cout << “The Heap is Empty” << endl;

    else {
        cout << “The root nodes of Heap are: ” << endl;
        do {
            cout << ptr->key;
            ptr = ptr->right;
            if (ptr != mini) {
                cout << “—>”;
            }
        } while (ptr != mini && ptr->right != NULL);
        cout << endl
             << "The heap has " << no_of_nodes << " nodes"
<< endl
             << endl;
    }
}

// Driver code
int main()
{
    // A heap is created and 3 nodes inserted into it
    cout << "Create the initial heap" << endl;
    insertion(5);
    insertion(2);
    insertion(8);

    // Now display the root list of the heap
    display();

    // Now extract the minimum value node from the heap
    cout << "Extracting min" << endl;
    Extract_min();
    display();

    // Now decrease the value of node '8' to '7'
    cout << “Decrease value of 8 to 7” << endl;
    Find(mini, 8, 7);
    display();

    // Now delete the node ‘7’
    cout << “Delete the node 7” << endl;
   Deletion(7);
    display();

    return 0;
}
```

Output:

```
Creating an initial heap
The root nodes of Heap are:
2—>5—>8
The heap has 3 nodes

Extracting min
The root nodes of Heap are:
5
The heap has 2 nodes

Decrease value of 8 to 7
The root nodes of Heap are:
5
The heap has 2 nodes
```

```
Delete the node 7
Key Deleted
The root nodes of Heap are:
5
The heap has 1 nodes
```

Leftist Heap

Leftist heaps, otherwise known as leftist trees, are priority queues, and they are implemented with a binary heap variation. All the nodes have an s value (or distance or rank), which is the distance to the nearest leaf. However, where a binary heap is always a complete self-balancing binary tree, the leftist tree can often be completely unbalanced.

Below you can see the leftist heap time complexities:

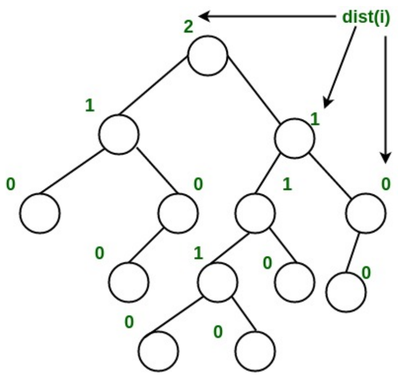
1. **Get Min**: O(1), which is the same as the binomial and binary heaps
2. **Delete Min**: O(Logn), which is the same as the binomial and binary heaps
3. **Insert**: O(Logn), which is O(Logn) in the binary heap, and o(1) in the binomial heap. The worst case is O(Logn)
4. **Merge**: O(Logn), which is O(Logn) in the binomial heap.

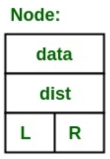
Leftist trees or heaps are binary trees with the following properties:

1. The normal Min Heap property is key(i) >= key(parent(i))
2. The left side is heavier – dist(right(i)) <= dist(left(i)). In this, dist(i) indicates how many edges there are on the shortest path between node i and an extended binary tree's leaf node. This representation considers a null child as a leaf or external node. The shortest path leading to a descendent external node will always go through the child on the right. All subtrees are leftist trees and dist(i) = 1 + dist(right(i))

Example:

The tree below shows each node's distance calculated using the above-mentioned procedure. The node on the absolute right has a rank of 0 because its right subtree is null and its parent's distance is 1 by dist(i) = 1 + dist(right)i)). Each node follows the same procedure, calculating its s-value (rank).





From the second property above, there are two conclusions we can draw:

1. The shortest path between a root and a leaf is the path between the root and the rightmost leaf
2. If there are x nodes on the path to the rightmost leaf, the leftist heap has a minimum of 2x – 1 nodes. This means the path to the rightmost leaf has a length of O(Logn), where the leftist heap has n nodes.

Operations

1. **merge() –** the primary operation
2. **deleteMin() or extractMin()** – the root is removed and merge() is called for the right and left subtrees
3. **insert()** – used to create leftist trees with single keys (the key to be inserted). The merge() operation is called for the specified tree and the tree with a single node.

Let’s look at the merge() operation in a bit more detail.

The idea behind this operation is that the right subtree from a tree is merged with another tree. These are the abstract steps:

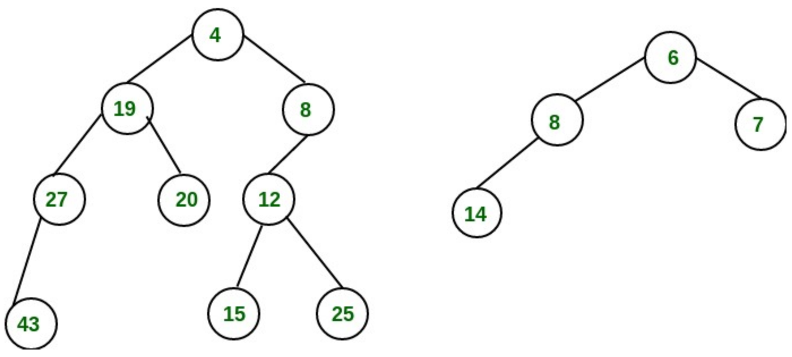
1. The root with the smallest value is set as the new root
2. The left subtree is hung on the left
3. The right subtree is recursively merged with the other tree
4. Before the recursion is returned from:
   - dist() is updated in the merged root
   - the left and right subtrees below the root are swapped if required to ensure the merged result retains its leftist property

Here are the detailed steps:

1. The roots of both heaps are compared
2. The smaller key is pushed into an empty stack, and then we move on to the smaller key's right child
3. The two keys are compared recursively, and we move on, pushing the smaller key to the stack and moving the key's right child
4. These steps are repeated until we reach a null node
5. The last processed node is made into the right child for the top-most node in the stack. If the leftist heap properties have been violated, the right child is converted into a leftist heap.
6. Continue to pop the elements recursively from the stack and convert them into the right child of the new top-most node stack.

Example

Look at the leftist heaps below:



When you merge these into one leftist heap, the leftist heap property is violated by the subtree at node 7. In that case, it is swapped with the left child, and the leftist heap property is retained.

Next, it is converted into a leftist heap, and the process is repeated.

This algorithm's worst-case time complexity is O)Logn), where n indicates how many nodes are in the leftist heap.

Below is a C++ implementation of a leftist heap:

```
//C++ program for leftist heap / leftist tree
#include <bits/stdc++.h>
using namespace std;

// Node Class Declaration
class LeftistNode
{
public:
    int element;
    LeftistNode *left;
    LeftistNode *right;
    int dist;
    LeftistNode(int & element, LeftistNode *lt = NULL,
                LeftistNode *rt = NULL, int np = 0)
    {
       this->element = element;
       right = rt;
       left = lt,
       dist = np;
    }
};

//Class Declaration
class LeftistHeap
{
public:
    LeftistHeap();
    LeftistHeap(LeftistHeap &rhs);
    ~LeftistHeap();
    bool isEmpty();
    bool isFull();
    int &findMin();
    void Insert(int &x);
    void deleteMin();
    void deleteMin(int &minItem);
    void makeEmpty();
    void Merge(LeftistHeap &rhs);
    LeftistHeap & operator =(LeftistHeap &rhs);
private:
    LeftistNode *root;
    LeftistNode *Merge(LeftistNode *h1,
                    LeftistNode *h2);
    LeftistNode *Merge1(LeftistNode *h1,
                     LeftistNode *h2);
    void swapChildren(LeftistNode * t);
    void reclaimMemory(LeftistNode * t);
    LeftistNode *clone(LeftistNode *t);
};

// Construct the leftist heap
LeftistHeap::LeftistHeap()
{
    root = NULL;
}
```

```
// Copy constructor.
LeftistHeap::LeftistHeap(LeftistHeap &rhs)
{
    root = NULL;
    *this = rhs;
}

// Destruct the leftist heap
LeftistHeap::~LeftistHeap()
{
    makeEmpty( );
}

/* Merge RHS into the priority queue.
RHS becomes empty. RHS must be different
from this.*/
void LeftistHeap::Merge(LeftistHeap &rhs)
{
    if (this == &rhs)
        return;
    root = Merge(root, rhs.root);
    rhs.root = NULL;
}

/* Internal method to merge two roots.
Deals with deviant cases and calls recursive Merge1.*/
```

```
LeftistNode *LeftistHeap::Merge(LeftistNode * h1,
                                LeftistNode * h2)
{
    if (h1 == NULL)
        return h2;
    if (h2 == NULL)
        return h1;
    if (h1->element < h2->element)
        return Merge1(h1, h2);
    else
        return Merge1(h2, h1);
}

/* Internal method to merge two roots.
Assumes trees are not empty, and h1's root contains
  the smallest item.*/
LeftistNode *LeftistHeap::Merge1(LeftistNode * h1,
                                 LeftistNode * h2)
{
    if (h1->left == NULL)
        h1->left = h2;
    else
    {
        h1->right = Merge(h1->right, h2);
        if (h1->left->dist < h1->right->dist)
            swapChildren(h1);
        h1->dist = h1->right->dist + 1;
    }
    return h1;
}

// Swaps t's two children.
void LeftistHeap::swapChildren(LeftistNode * t)
{
    LeftistNode *tmp = t->left;
    t->left = t->right;
    t->right = tmp;
}

/* Insert item x into the priority queue, maintaining
  heap order.*/
void LeftistHeap::Insert(int &x)
{
    root = Merge(new LeftistNode(x), root);
}

/* Find the smallest item in the priority queue.
Return the smallest item, or throw Underflow if empty.*/
int &LeftistHeap::findMin()
{
    return root->element;
}

/* Remove the smallest item from the priority queue.
```

```
Throws Underflow if empty.*/
void LeftistHeap::deleteMin()
{
    LeftistNode *oldRoot = root;
    root = Merge(root->left, root->right);
    delete oldRoot;
}

/* Remove the smallest item from the priority queue.
Pass the smallest item back, or throw Underflow if
empty.*/
void LeftistHeap::deleteMin(int &minItem)
{
    if (isEmpty())
    {
        cout<<"Heap is Empty"<<endl;
        return;
    }
    minItem = findMin();
    deleteMin();
}

/* Test if the priority queue is logically empty.
Returns true if empty, false otherwise*/
bool LeftistHeap::isEmpty()
{
    return root == NULL;
}

/* Test if the priority queue is logically full.
Returns false in this implementation.*/
bool LeftistHeap::isFull()
{
    return false;
}

// Make the priority queue logically empty
void LeftistHeap::makeEmpty()
{
    reclaimMemory(root);
    root = NULL;
}

// Deep copy
LeftistHeap &LeftistHeap::operator =(LeftistHeap & rhs)
{
    if (this != &rhs)
    {
        makeEmpty();
        root = clone(rhs.root);
    }
    return *this;
}
```

```
// Internal method to make the tree empty.
void LeftistHeap::reclaimMemory(LeftistNode * t)
{
    if (t != NULL)
    {
        reclaimMemory(t->left);
        reclaimMemory(t->right);
        delete t;
    }
}

// Internal method to clone the subtree.
LeftistNode *LeftistHeap::clone(LeftistNode * t)
{
    if (t == NULL)
        return NULL;
    else
        return new LeftistNode(t->element, clone(t->left),
                               clone(t->right), t->dist);
}

//Driver program
int main()
{
    LeftistHeap h;
    LeftistHeap h1;
    LeftistHeap h2;
```

```
        int x;
        int arr[]= {1, 5, 7, 10, 15};
        int arr1[]= {22, 75};

        h.Insert(arr[0]);
        h.Insert(arr[1]);
        h.Insert(arr[2]);
        h.Insert(arr[3]);
        h.Insert(arr[4]);
        h1.Insert(arr1[0]);
        h1.Insert(arr1[1]);

        h.deleteMin(x);
        cout<< x <<endl;

        h1.deleteMin(x);
        cout<< x <<endl;

        h.Merge(h1);
        h2 = h;

        h2.deleteMin(x);
        cout<< x << endl;

        return 0;
    }
```
Output:

1

22

5

K-ary Heap

The k-ary heap is a binary heap generalization (k=2) where every node has k children rather than just 2. As with the binary heap, the k-ary heap has two properties:

1. A k-ary heap is an almost complete binary tree, where all the levels have the maximum number of nodes, except for the final one – this is filled from left to right.
2. Like the binary heap, we can divide a k-ary heap into two categories:
   a. Max k-ary heap – the key at the root is more than all the descendants. The same is true recursively for every node
   b. Min k-ary heap – the key at the root is less than the descendants. The same is true recursively for every node

Examples:

In a 3-ary max-heap, the maximum of all the nodes is the root node:

```
        10

     /   |  \

  7      9     8

/|\  /

4  6  5 7
```

In a 3-ary min-heap, the minimum of all the nodes is the root node:

```
        10

      /  | \

    12    11  13

  /|\

14 15 18
```

A complete k-ary tree containing n nodes has a height given by logkn.

K-Ary Heap Applications

1. When using it in a priority queue implementation, the k-ary heap has a faster decrease_key() operation than the binary heap. The binary heap time complexity is O(Log2n), where the k-ary heap is O(Logkn). However, it does cause the extractMin() operation's time complexity to increase. The binary heap does it in time complexity of O(Log2n) while the k-ary heap's complexity is O(k log ln). This does mean the k-ary heap is efficient where the decrease priority operations in an algorithm are more common than the extractMin() operation. An example of this is the Prim and Djikstra algorithms.
2. The memory cache behavior in the k-ary heap is much better than in a binary heap. This means that, in practice, the k-ary heap runs faster, but their worst-case running times for the delete() and extractMin() operations are much larger, with both of them being O(K Log kn)

Implementation

Assuming we have an array with O-based indexing, the k-ary heap is represented by the array in such a way that the following are considered for any node:

- For the node at index i (unless it is a root node), the parent is at index (i-1)/k
- For the node at index i, the children are located at the (k*i)+1, (k*i)+2, …, (k*i)+k indices
- In a heap of size n, the last non-leaf node is at (n-2)/k

buildHeap():

This function takes an input array and builds a heap. Starting from the last non-leaf node, the function runs a loop to the root node. The restoreDown function, also called maHeapify, is called for every index where the passed index is stored in the right place in the heap. The node is moved down the heap, building the heap from the bottom up.

So, why does the loop need to start from the last non-leaf node?

The simple answer is every node that comes after that is a leaf node. Because they have no children, they satisfy the heap property trivially and, as such, are already max-heap roots.

restoreDown() (or maxHeapify):

This function maintains the heap property, running a loop through the node's children to find the maximum. It then compares the maximum with its own value, swapping where max(value of all the children) > (the node's value). This is repeated until the node is back to its original position within the k-ary heap.

extractMax():

This function extracts the root node. In a k-ary heap, the largest element is stored in the heap's root. The root node is returned, the last node copied to the first, and restoreDown is called on the first node, ensuring the heap's property is maintained.

insert():

This function is used to insert a node into the k-ary heap. The node is inserted at the last position, and restoreUp() is called on the given index, putting the node back to its rightful position in the k-ary heap. The restoreUp() function compares a specified node iteratively with its parent because the parent in a max heap will always be equal to or greater than its children nodes. The node and parent can only be swapped when the node's key is greater than the parent.

In the following C++ implementation, we put all of this together:

```
// C++ program to demonstrate all the operations of

// k-ary Heap
```

```
#include<bits/stdc++.h>

using namespace std;

// function to heapify (or restore the max- heap
// property). This is used to build a k-ary heap
// and in extractMin()
// att[] — Array that stores heap
// len   — Size of array
// index — index of element to be restored
//         (or heapified)
void restoreDown(int arr[], int len, int index,
                                        int k)
{
    // child array to store indexes of all
    // the children of given node
    int child[k+1];

    while (1)
    {
        // child[i]=-1 if the node is a leaf
        // children (no children)
        for (int i=1; i<=k; i++)
            child[i] = ((k*index + i) < len) ?
                    (k*index + i) : -1;

        // max_child stores the maximum child and
        // max_child_index holds its index
        int max_child = -1, max_child_index ;

        // loop to find the maximum of all
        // the children of a given node
        for (int i=1; i<=k; i++)
        {
            if (child[i] != -1 &&
                arr[child[i]] > max_child)
            {
                max_child_index = child[i];
                max_child = arr[child[i]];
            }
        }

        // leaf node
        if (max_child == -1)
            break;

        // only swap if the max_child_index key
        // is greater than the node's key
        if (arr[index] < arr[max_child_index])
            swap(arr[index], arr[max_child_index]);

        index = max_child_index;
    }
}
```

```
// Restores a given node up in the heap. This is used
// in decreaseKey() and insert()
void restoreUp(int arr[], int index, int k)
{
    // parent stores the index of the parent variable
    // of the node
    int parent = (index-1)/k;

    // Loop should run only until the root node in case the
    // element inserted is the maximum. restoreUp will
    // send it to the root node
    while (parent>=0)
    {
        if (arr[index] > arr[parent])
        {
            swap(arr[index], arr[parent]);
           index = parent;
            parent = (index -1)/k;
        }

        // the node has been restored in the correct position
        else
            break;
    }
}
```

```
// Function to build a heap of arr[0..n-1] and alue of k.
void buildHeap(int arr[], int n, int k)
{
    // all internal nodes are heapified starting from last
    // non-leaf node up to the root node
    // and calling restoreDown on each
    for (int i= (n-1)/k; i>=0; i—)
        restoreDown(arr, n, i, k);
}

// Function to insert a value in a heap. The parameters are
// the array, size of the heap, value k, and the element to
// be inserted
void insert(int arr[], int* n, int k, int elem)
{
    // Put the new element in the last position
    arr[*n] = elem;

    // Increase heap size by 1
    *n = *n+1;

    // Call restoreUp on the last index
    restoreUp(arr, *n-1, k);
}

// Function to returns the key of the heap's root node
// and restores the heap property
// for the remaining nodes
int extractMax(int arr[], int* n, int k)
{
    // Stores the key of root node to be returned
    int max = arr[0];

    // Copy the last node’s key to the root node
    arr[0] = arr[*n-1];

    // Decrease heap size by 1
    *n = *n-1;

    // Call restoreDown on the root node to restore
    // it to the correct position in the heap
    restoreDown(arr, *n, 0, k);

    return max;
}

// Driver program
int main()
{
    const int capacity = 100;
    int arr[capacity] = {4, 5, 6, 7, 8, 9, 10};
    int n = 7;
    int k = 3;
    buildHeap(arr, n, k);

    printf("Built Heap : \n");
    for (int i=0; i<n; i++)
       printf("%d ", arr[i]);

    int element = 3;
    insert(arr, &n, k, element);

    printf("\n\nHeap after insertion of %d: \n",
          element);
    for (int i=0; i<n; i++)
       printf("%d ", arr[i]);

    printf("\n\nExtracted max is %d",
                extractMax(arr, &n, k));

    printf("\n\nHeap after extract max: \n");
    for (int i=0; i<n; i++)
       printf("%d ", arr[i]);

    return 0;
}
```

Output

```
Built Heap :
10 9 6 7 8 4 5
```

```
Heap after insertion of 3:
10 9 6 7 8 4 5 3

Extracted max is 10

Heap after extract max:
9 8 6 7 3 4 5
```

Time Complexity

- Where a k-ary heap has n nodes, logkn is the given heap's maximum height. The restoreUp() function is run for no more than logkn times because the node is moved up a level at each iteration or down a level when restoreDown() is used.
- The restoreDown() function recursively calls itself for k children, giving it O(klogkn) time complexity.
- The decreaseKey() and insert() operations will only call restoreUp once, giving them O(klogkn) time complexity.
- The extractMax() function calls restoreDown() once, giving it O(kogkn) time complexity.
- The build heap has O(n) time complexity, much like the binary heap.

Heapsort/Iterative HeapSort

Before we examine the iterative heapsort data structure, we need to understand what the basic heapsort is. It is a sorting technique that uses comparisons and is based on the binary heap structure. It is a little like the selection sort data structure where the minimum element is found and placed at the top. Then, the process is repeated for the rest of the elements.

We know that the binary heap is a complete binary tree that stores items in special orders, such as max heap, where the parent node's value is greater than the values in the pair of children's nodes, or min-heap, where the parent node's value is less. Binary heaps are represented by arrays or binary trees.

Arrays can easily represent the binary heap and are one of the most efficient methods in terms of space. If you store a parent node at index I, we can calculate the left child with 2 * I and the right child with 2 * I +2.

Heapsort Algorithm

The following is the algorithm used for the heapsort data structure:

1. Use the input data to build a max heap.
2. The largest item should be stored in the heap's root. Replace this with the heap's last item and then reduce the heap size by 1. Lastly, the tree's root is heapified.
3. Repeat the previous step until the heap size falls to or below 1.

Building the Heap

We can only apply the heapify procedure to a node whose children have been heapified, which means heapification can only be done from the bottom up. Here's an example to make that clearer:

Input data: 4, 10, 3, 5, 1

```
        4(0)
```

```
       /   \
     10(1)   3(2)
    /   \
5(3)    1(4)
```

The bracketed numbers represent the indices in the array's data representation.

Apply heapify to index 1:

```
        4(0)
       /   \
    10(1)    3(2)
    /   \
5(3)    1(4)
```

Apply heapify to index 0:

```
       10(0)
       /  \
     5(1)  3(2)
    /   \
4(3)    1(4)
```

The heap is built top-down by the heapify process recursively calling itself.

Below is a C++ implementation of heapsort:

```
// C++ program to implement Heapsort
#include <iostream>

using namespace std;

// heapify a subtree rooted with a node i which is
// an index in arr[]. n is the size of heap
```

```
void heapify(int arr[], int n, int i)
{
    int largest = i; // Initialize largest as root
    int l = 2 * i + 1; // left = 2*i + 1
    int r = 2 * i + 2; // right = 2*i + 2

    // If left child is larger than root
    if (l < n && arr[l] > arr[largest])
        largest = l;

    // If right child is larger than largest so far
    if (r < n && arr[r] > arr[largest])
        largest = r;

    // If largest is not root
    if (largest != i) {
        swap(arr[i], arr[largest]);

        // Heapify the affected sub-tree recursively
        heapify(arr, n, largest);
    }
}

// main function to do a heapsort
void heapSort(int arr[], int n)
{
    // Build the heap (rearrange array)
    for (int i = n / 2 - 1; i >= 0; i—)
        heapify(arr, n, i);

    // Extract the elements from the heap one by one
    for (int i = n - 1; i > 0; i—) {
        // Move the current root to end
        swap(arr[0], arr[i]);

        // call max heapify on the reduced heap
        heapify(arr, i, 0);
    }
}

/* utility function that prints an array of size n */
void printArray(int arr[], int n)
{
    for (int i = 0; i < n; ++i)
        cout << arr[i] << ” “;
    cout << “\n”;
}

// Driver code
int main()
{
    int arr[] = { 12, 11, 13, 5, 6, 7 };
    int n = sizeof(arr) / sizeof(arr[0]);
```

```
        heapSort(arr, n);


        cout << "Sorted array is \n";
        printArray(arr, n);
    }
```

Output

```
Sorted array is
5 6 7 11 12 13
```

Time Complexity

Heapify has O(Logn) time complexity, createAndBuildHeap() has O(n) time complexity, while heapsort has an overall time complexity of O(nLogn).

Advantages

- **Efficiency** – The time needed to perform the heapsort logarithmically increases, while many other algorithms tend to slow down exponentially as the number of items needing sorting increases. That makes this an efficient sorting algorithm.
- **Memory Usage** – This is minimal. No additional memory space is required aside from what is needed for the initial sorting list.
- **Simplicity** – It is one of the simpler sorting algorithms to understand because advanced computer science concepts, such as recursion, are not used.

Here's a more in-depth example of using the heapsort algorithm. If you remember, the max heap is built first, and the root element is swapped with the last element. The heap property is maintained every time to ensure the list is sorted

```
Input :  10 20 15 17 9 21
Output : 9 10 15 17 20 21
```

Input: 12 11 13 5 6 7 15 5 19

Output: 5 5 6 7 11 12 13 15 19

In the first example, the max heap needs to be built, so we will begin from 20, a child node, and check for the parent node. 10 is smaller, so the two are swapped:

20 10 15 17 9 21

The child node 17 is now greater than the parent 10, so these are swapped:

20 17 15 10 9 21

The child node 21 is now greater than the parent node 15, so these are swapped:

20 17 21 10 9 15

And, again, the child node 21 is greater than the parent 20, so we swap them:

21 17 20 10 9 15

This is the max heap.

Next, sorting must be applied. The first and last elements are swapped, and the max heap property must be maintained. After the first swap, we have:

15 17 20 10 9 21

Clearly, the max heap property has been violated, so it needs to be maintained. This is the order:

20 17 15 10 9 21

17 10 15 9 20 21

15 10 9 17 20 21

10 9 15 17 20 21

9 10 15 17 20 21

The underlined numbers are the sorted numbers.

Here is the C++ implementation:

```
// C++ program for implementation
// of the Iterative Heap Sort
#include <bits/stdc++.h>
using namespace std;

// function to build Max Heap where thevalue
// of each child is always smaller
// than value of their parent
void buildMaxHeap(int arr[], int n)
{
    for (int i = 1; i < n; i++)
    {
        // if the child is bigger than the parent
        if (arr[i] > arr[(i - 1) / 2])
        {
            int j = i;

            // swap the child and parent until
            // the parent is smaller
            while (arr[j] > arr[(j - 1) / 2])
            {
                swap(arr[j], arr[(j - 1) / 2]);
                 j = (j - 1) / 2;
            }
        }
    }
}
```

```
void heapSort(int arr[], int n)
{
    buildMaxHeap(arr, n);

    for (int i = n - 1; i > 0; i—)
    {
       // swap the value of first indexed
       // with the last indexed
       swap(arr[0], arr[i]);

       // maintaining the heap property
       // after each swapping
       int j = 0, index;

       do
       {
          index = (2 * j + 1);

          // if the left child is smaller than
          // the right child point the index variable
          // to the right child
          if (arr[index] < arr[index + 1] &&
                         index < (i - 1))
             index++;

          // if the parent is smaller than the child
          // then swap the parent with the child
```

```
            // with the higher value
            if (arr[j] < arr[index] && index < i)
                swap(arr[j], arr[index]);

            j = index;

        } while (index < i);
    }
}

// Driver Code to test above
int main()
{
    int arr[] = {10, 20, 15, 17, 9, 21};
    int n = sizeof(arr) / sizeof(arr[0]);

    printf("Given array: ");
    for (int i = 0; i < n; i++)
        printf("%d ", arr[i]);

    printf("\n\n");

    heapSort(arr, n);

    // print array after sorting
    printf("Sorted array: ");
    for (int i = 0; i < n; i++)
```

```
        printf("%d ", arr[i]);


    return 0;
}
```

Output:

```
Given array: 10 20 15 17 9 21


Sorted array: 9 10 15 17 20 21
```

Time Complexity

The heapSort and buildMaxHeap functions both run in O(nLogn) time, so O(nLogn) is the overall time complexity.

Conclusion

Are you wondering why you should bother to study advanced data structures? I mean, what use do they really have in the real world? When you go for a data science or data engineering job interview, why do the interviewers insist on asking questions about them?

Many programmers, beginners, and experienced ones alike choose to avoid learning about algorithms and data structures. This is partly because they are quite complicated and partly because they don't see the need for them in real life.

However, learning data structures and algorithms allows you to see real-world problems and their solutions in terms of code. They teach you how to take a problem and break it down into several smaller pieces. You think about how each piece fits together to provide the final solution and can easily see what you need to add along the way to make it more efficient.

This guide has taken you through some of the more advanced data structures used in algorithms and given you some idea of how they work and where they will work the best. It is an advanced guide, and you are not expected to understand everything right away. However, you should use this guide as your go-to when you want to double-check that a data structure will fit your situation or when you want to see if there is a better way of doing something.

Thank you for taking the time to read it. I hope you found it helpful and now feel you can move on in your data journey.

References

"15-121 Priority Queues and Heaps." Www.cs.cmu.edu, www.cs.cmu.edu/~rdriley/121/notes/heaps.html.

"Advanced Data Structures." GeeksforGeeks, www.geeksforgeeks.org/advanced-data-structures/.

"Bloom Filters - Introduction and Python Implementation." GeeksforGeeks, 17 Apr. 2017, www.geeksforgeeks.org/bloom-filters-introduction-and-python-implementation/.

"DAA Red Black Tree - Javatpoint." Www.javatpoint.com, www.javatpoint.com/daa-red-black-tree.

"Different Self Balancing Binary Trees." OpenGenus IQ: Computing Expertise & Legacy, 20 Aug. 2020, iq.opengenus.org/different-self-balancing-binary-trees/.

"Disjoint Set Data Structure - Javatpoint." Www.javatpoint.com, www.javatpoint.com/disjoint-set-data-structure.

"Disjoint Sets - Disjoint Sets Union, Pairwise Disjoint Sets." Cuemath, www.cuemath.com/algebra/disjoint-sets/.

"Disjoint Sets - Explanation and Examples." The Story of Mathematics - a History of Mathematical Thought from Ancient Times to the Modern Day, www.storyofmathematics.com/disjoint-sets.

"Doubly Linked List - Javatpoint." Www.javatpoint.com, www.javatpoint.com/doubly-linked-list.

"Fenwick Tree - Competitive Programming Algorithms." Cp-Algorithms.com, cp-algorithms.com/data_structures/fenwick.html.

"N-Ary Tree Data Structure - Studytonight." Www.studytonight.com, www.studytonight.com/advanced-data-structures/nary-tree.

"Persistent Data Structures." GeeksforGeeks, 27 Mar. 2016, www.geeksforgeeks.org/persistent-data-structures/.

"Segment Trees Tutorials & Notes | Data Structures." HackerEarth, www.hackerearth.com/practice/data-structures/advanced-data-structures/segment-trees/tutorial/.

"Self-Balancing-Binary-Search-Trees (Comparisons)." GeeksforGeeks, 6 June 2018, www.geeksforgeeks.org/self-balancing-binary-search-trees-comparisons/.

Talbot, Jamie. "What Are Bloom Filters?" 3 Min Read, 3 min read, 15 July 2015, blog.medium.com/what-are-bloom-filters-1ec2a50c68ff.

"Treap (a Randomized Binary Search Tree)." GeeksforGeeks, 22 Oct. 2015, www.geeksforgeeks.org/treap-a-randomized-binary-search-tree/.

"Trie Data Structure - Javatpoint." Www.javatpoint.com, www.javatpoint.com/trie-data-structure.

"Unrolled Linked List | Brilliant Math & Science Wiki." Brilliant.org, brilliant.org/wiki/unrolled-linked-list/.

"Why Data Structures and Algorithms Are Important to Learn?" GeeksforGeeks, 13 Dec. 2019, www.geeksforgeeks.org/why-data-structures-and-algorithms-are-

important-to-learn/