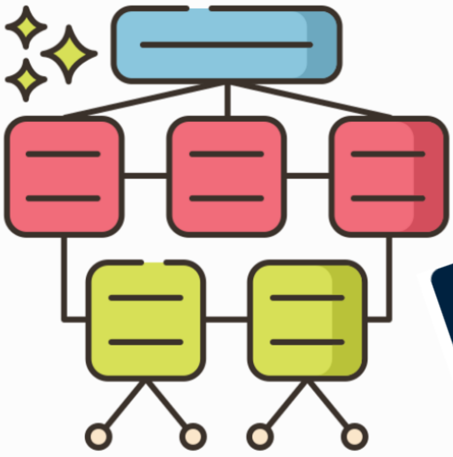
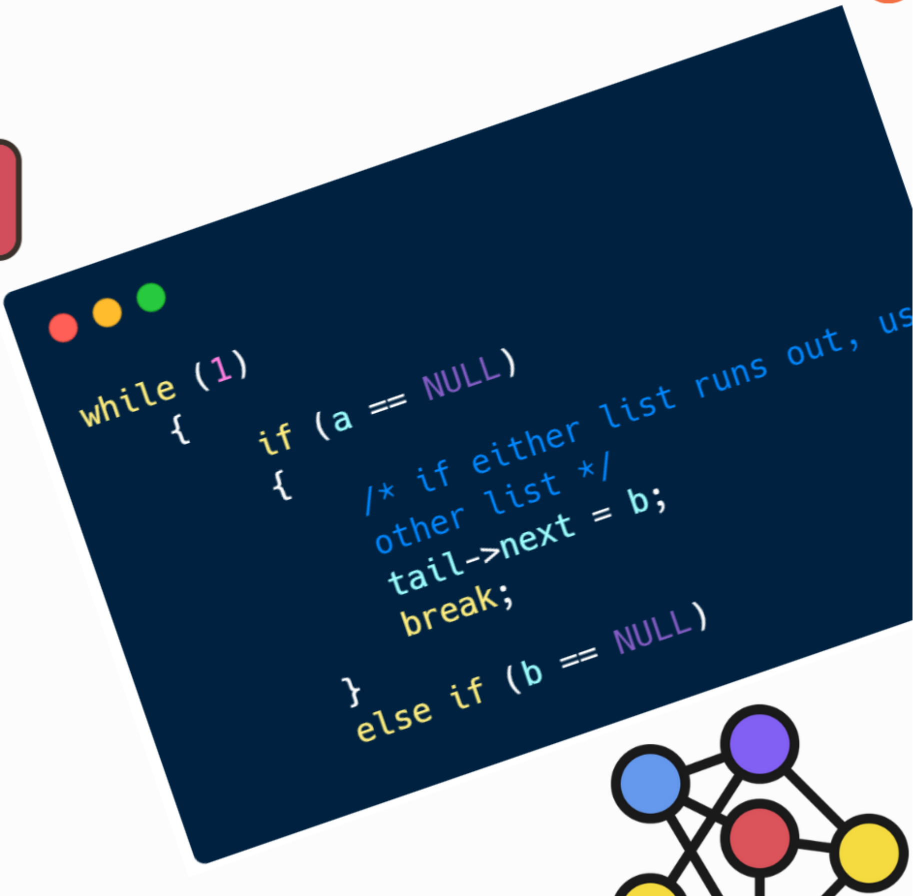
ALL IN ONE DATA STRUCTURES GUIDEBOOK





```
while (1)
{
    if (a == NULL)
    {
        /* if either list runs out, us
        other list */
        tail->next = b;
        break;
    }
    else if (b == NULL)
```

AARYAN R LONDHE

Table of Contents

Introduction

Introduction to Data Structures

A data structure is a model where data is organized, managed and stored in a format that enables efficient access and modification of data. There are various types of data structures commonly available. It is up to the programmer to choose which data structure to use depending on the data.

The choice of a particular one can be considered based on the following points:

1. It must be able to process the data efficiently when necessary.
2. It must be able to represent the inherent relationship of the data in the real world

Why study Data Structures?

"Bad programmers worry about the code. Good programmers worry about data structures and their relationships."

~ *Linus Torvalds*

Time and energy are both required to process any instruction. Every CPU cycle that is saved will have an effect on both the time and energy consumed and can be put to better use in processing other instructions.

A program built using improper data structures will be therefore inefficient or unnecessarily complex. It is necessary to have a good knowledge of data structures and understand where to use the best one. The study includes the description, implementation and quantitative performance analysis of the data structure.

Concept of a Data Type

Primitive Data Types

A primitive data type is one that is inbuilt into the programming language for defining the most basic types of data. These may be different for the various programming languages available. For example, the C programming language has inbuilt support for characters (char), integers (int, long) and real numbers (float, double).

User-Defined Data Type

User-defined data type, as the name suggests is the one that the user defines as per the requirements of the data to be stored. Most programming languages provide support for creating user-defined data types. For example, C provides support through structures (struct), unions (union) and enumerations (enum).

Abstract Data Type (ADT)

Abstract Data Types are defined by its behaviour from the point of view of the user of the data. It defines it in terms of possible values, operations on data, and the behaviour of these operations.

The definition of ADT only mentions what operations are to be performed but not how these operations will be implemented. That is, it does not specify how the data is being handled under the hood. This concept is known as abstraction.

For eg. The user of the stack data structure only knows about the push and pop operations in a stack. They do not care how the push operation interacts with the memory to store the data. They only expect it to store it in the way specified.

Common Operations in a Data Structure

A data structure is only useful when you can perform operations on it, right? These are the basic operations that should be able to be performed on every data structure.

Access

This operation handles how the elements currently stored in the structure can be accessed.

Search

This operation handles finding the location of a given element of a given structure.

Insertion

This operation specifies how new elements are to be added to the structure.

Deletion

This operation specifies how existing elements can be removed from the structure.

Classification of Data Structure

A data structure can be broadly classified into 2 types:

Linear Data Structures

A linear data structure's elements form a sequence. Every element in the structure has some element before and after it. Examples of linear structures are:

Arrays

An array holds a fixed number of similar elements that are stored under one name. These elements are stored in contagious memory locations . The elements of an array can be accessed using one identifier.

Linked Lists

A linked list is a linear data structure where each element is a separate object, known as a node . Each node contains some data and points to the next node in the structure, forming a sequence . .

Stacks

Stacks are a type of linear data structures that store data in an order known as the **Last In First Out (LIFO)** order. This property is helpful in certain programming cases where the data needs to be ordered.

Queues

Queues are a type of linear data structures that store data in an order known as the **First In First Out (FIFO)** order. This property is helpful in certain programming cases where the data needs to be ordered.

Non-Linear Data Structures

A non-linear data structure's elements do not form a sequence. Every element may not have a unique element before and after it.

Trees

A tree is a data structure that simulates a hierarchical tree, with a root value and the children as the subtrees, represented by a set of linked nodes.

Heaps

A heap is a specialized tree-based data structure that satisfies the **heap** property.

Graphs

A graph data structure is used to represent relations between pairs of objects. It consists of **nodes** (known as vertices) that are connected through **links** (known as edges). The relationship between the nodes can be used to model the relation between the objects in the graph.

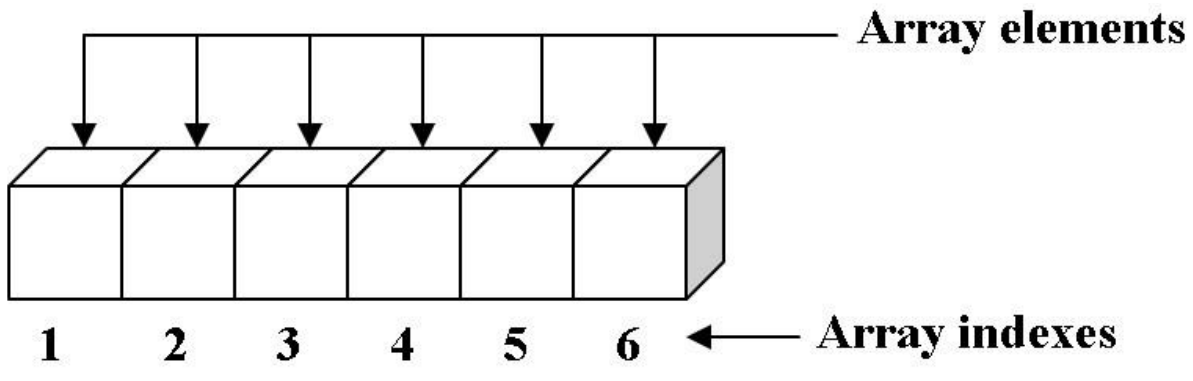
Hash Tables

A Hash Table is a data structure where data is stored in an associative manner. The data is mapped to array positions by a hash function that generates a unique value from each key. The value stored in a hash table can then be searched in **O(1)** time using the same hash function which generates an address from the key.

Arrays

What is Arrays?

An array holds a fixed number of similar elements that are stored under **one name**. These elements are stored in **contagious memory locations**. It is one of the simplest data structures. Most modern programming languages have arrays built-in by default.



One-dimensional array with six elements

Why use arrays over a bunch of variables?

The reason why we use arrays is that every element can be accessed by its index value. This has several advantages over storing a bunch of variables.

For example: Consider we have to implement a system to store the age of all employees in an office. There is the traditional way with variables.

One can create a variable for each employee in the office. Let's say the office has only 3 employees. Fairly easy right? Just declare 3 variables: emp1_age, emp2_age and emp3_age.

When new recruitments come in, we sit down to create more variables. Maintaining a system like this gets tedious. Imagine one new employee and the whole system code has to be modified.

Accessing each variable would also be a headache. It is stupid to sum 20 variables by hand to calculate the average age of the employees.

An array data structure tries to solve these problems.

One of the properties of arrays is that it holds the same kind of data under one name.

For this example, the array can hold all the ages of the employees under one name, like employees_age. These are all of the *integer* type.

The second property of arrays is that it stores each element in a continuous block which can be accessed using its index.

Every employee's age can be accessed by iterating through the indices of the array. This can be used to easily access all values serially by looping through them. The function to calculate average becomes much easier to implement as the name of the array is constant and only the index is changing.

Let's see how an array is declared and used.

Declaring a One Dimensional Array

An array has to be declared before it can be used. In C, declaring an array means specifying the following:

- Data Type: This is the kind of values that the array will store. This can be characters, integers, floating points or any legal data type.

- Name: The variable name used to identify the array and interact with it.
- Size: The size of the array, which specifies the maximum number of values that the array will store.

Syntax Used

An array can be declared in C by using the following syntax:

```
type name[size];
```

For example, an array of marks of a class of 100 students can be created using:

```
1. int marks[100];
```

There are 2 ways to assign elements to an array:

Assigning Values While Initialization

The values of the elements can be assigned while declaring the array. If some of the values are not explicitly defined, they are set to 0.

```
1. int marks[10] = {5, 10, 20, 30, 40, 60};
```

Assigning values after initialization

By default, an array is created whenever memory is available at any random location. We do not know what information that random location of memory will contain, as any other program could have used that memory previously.

If array elements are not initialized while creation, then accessing them directly they would result in such garbage values.

Therefore, it is always recommended to empty the elements or assign values to it if a calculation is to be performed on the array.

```
int ages[10];

// accessing array without assigning elements first
for(int i = 0; i < 10; i++)
  printf("\n arr[%d] = %d", i, ages[i]);
```

Traversing The Array

Each element of the array can be accessed using its index. The indexing in an array generally starts with 0, which means that the first element is at the 0th index. Subsequently, the last element of the array would be at the (n-1)th index. This is known as 0-based indexing.

The indexing of the array may also be different by using any other base. These are known as n-based indexing.

Accessing all the elements is possible by using a simple for-loop going through all the indices in the array.

```
for(int i = 0; i < arraySize; i++)
  printf("\n arr[%d] = %d", i, arr[i]);
```

Example: Values are first being assigned and then displayed from the array.

```
int id[10];

// assigning values using a loop

for (int i = 0; i < 10; i++) {
  printf("\nEnter an id: ");
  scanf("%d", &id[i]);
}

// displaying the entered ids

for (int i = 0; i < 10; i++) {
  printf("\n id[%d] = %d", i, id[i]);
}
```

Maintaining the order of an array while inserting or deleting requires manipulating the others already present in the array. This is one of the disadvantages, as such operations can be costly on larger arrays.

Inserting An Element In The Array

At the end

Inserting an element at the end of the array is easy provided the array has enough space for the new element. The index of the last element of the array is found out and the new element is inserted at the index + 1 position.

At any other position

An element can be inserted in between at any position by shifting all elements from that position to the back of the array. The element to be inserted is then inserted at the required position.

```
void insert_position(int arr[]) {
    int i = 0, pos, num;
    printf("Enter the number to be inserted : ");
    scanf("%d", &num);
    printf("Enter position at which the number is to be added :");
    scanf("%d", &pos);
    for (i = n-1; i>= pos; i--)
        arr[i+1] = arr[i];
    arr[pos] = num;
     n = n + 1;  // increase total number of used positions
    display_array(arr);
}
```

Deleting An Element From The Array

At the end

Deleting an element at the end of the array is equally easy provided there is some element to begin with (not an empty array). The index of the last element is found out and this element is deleted.

At any other position

An element can be deleted at any index by deleting the element at that position and then moving up all the elements from the back of the array to the front to fill up the position of the deleted element.

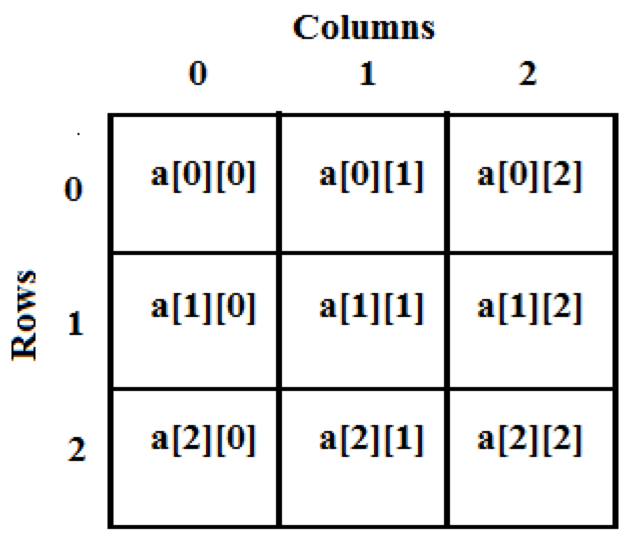
```
void delete_position(int arr[]) {
    int i, pos;
    printf("\nEnter the position where the number has to be deleted: ");
    scanf("%d", &pos);
    for (i = pos; i < n-1; i++)
        arr[i] = arr[i+1];
    n = n - 1;  // decrease total number of used positions
    display_array(arr);
}
```

Multi-Dimensional Arrays

An array may have more than one dimension to represent data. These are known as multidimensional arrays. The elements in these arrays are accessed using multiple indices.

Two Dimensional Array

A two dimensional array can be considered as an array within an array. It can be visualised as a table, having a row and a column. Each item in the table can be accessed using 2 indices corresponding to the row and column.



		Columns		
		0	1	2
Rows	0	a[0][0]	a[0][1]	a[0][2]
	1	a[1][0]	a[1][1]	a[1][2]
	2	a[2][0]	a[2][1]	a[2][2]

Example of a 2D array and its indices

A 2D array is declared using 2 parameters:

```
type name[max_size_x][max_size_y]
```

The max_size_x and max_size_y are the max values each dimension can store.

Three Dimensional Array

A three dimensional array similarly can be visualised as a cube. Each item can be accessed using 3 indices corresponding to the 3D position.

Example: Every block of a Rubik's Cube can be represented by a three dimensional array of size 3 x 3 x 3.

A 3D array is declared using 2 parameters:

```
type name[max_size_x][max_size_y][max_size_z];
```

The max_size_x, max_size_y and max_size_z are the max values each dimension can store.

Memory Allocation in Arrays

Since an array stores all its data elements in consecutive memory locations, storing just the base address, that is, the address of the first element is sufficient. The address of other elements can be then calculated using the base address.

For One Dimensional Array

A simple formula consisting of the size of the element and the lower bound is used.

```
A[i] = base_address(A) + size_of_element(i – lower_bound)
```

The *lower bound* is the smallest index in the array. Similarly, an *upper bound* is the largest index in the array. In C programming, the lower bound value may be omitted as it is generally 0.

For Two Dimensional Array

The elements in a two dimensional array can be stored using 2 representations and their addresses can be calculated using the respective formulae.

Column Major Representation

In this form, the elements are stored column by column. *m* elements of the first column are stored in the first *m* locations, elements of the second column element are stored in the next *m* locations, and so on.

Address(A[I][J]) = base_address + width {number_of_rows (J – 1) + (I – 1)}

Row Major Formula

In this form, the elements are stored row by row. *n* elements of the first row are stored in the first *n* locations, elements of the second row elements are stored in the next *n* locations, and so on.

Address(A[I][J]) = base_address + width {number_of_cols (I – 1) + (J – 1)}

Time Complexity of Operations

Access

Any array element could be accessed directly through its index. Hence the access time is constant O(1).

Search

Searching for a given value through the array requires iterating through each element in the array until the element is found. This is assuming that linear search is used (which is the most basic type of search to find any element). This makes the search time O(n).

The other more efficient search algorithm, binary search could be used to search in O(log n) time but it requires the array to be sorted beforehand.

Insertion

Inserting an element in between 2 elements in an array involves shifting all the elements to the right by 1. This means that at most all the elements have to be shifted right (insertion at the beginning of the array), hence the complexity of the insert operation in O(n).

Deletion

Deleting an element in between 2 elements in an array involves shifting all the elements to the left by 1. This means that at most all the elements have to be shifted left (deletion at the beginning of the array), hence the complexity of the delete operation in O(n).

Space Required

An array only takes the space used to store the elements of the data type specified. This means that for storing n elements the space required is O(n).

Advantages of Arrays

Arrays have various advantages over other more complex data structures.

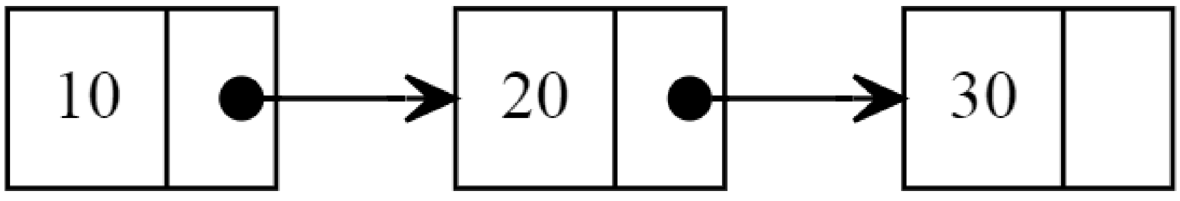
- Arrays allow for **random access** of elements. Each element in the array can be interacted with by directly accessing to its index.
- Arrays have **good cache locality**, which means the speed of execution of code may be significantly faster in some cases due to nature how arrays are stored.

Disadvantages of Arrays

- The size of an array is **fixed** once declared. This may be insufficient or more than required later on the program. In case the size is inefficient, it may be costly to move all the array elements to a new bigger array.
- Insertion and deletion of elements in the array so that it maintains a continuous order may be **expensive**, as one may have to relocate all the elements.

Linked List

A linked list is a linear data structure where each element is a separate object, known as a node. Each node contains some data and points to the next node in the structure, forming a sequence. The nodes may be at different memory locations, unlike arrays where all the elements are stored continuously.



Linked List

The linked list can be used to store data similar to arrays but with several more advantages.

Think of it as a friend circle. If person A knows person B and person B knows person C, person C could be reached from person A through this linked connection. Each person can be seen as a Node who knows the link to the next person.

Advantages over Arrays

The 2 advantages of a linked list over an array are:

- Not fixed in size: A linked list is not fixed in size. The memory locations to store the nodes are allocated dynamically when each node is created. There is no wastage of memory for unused locations. In comparison, an array can only be defined once of a specific size, and then further cannot be extended or shrunk down accordingly.

- Efficient Insertion and Deletion: A quick manipulation of the links between the nodes allows for a constant time taken for insertion and deletion. In contrast, one has to move over all the memory locations while dealing with arrays so that they are in order.

Disadvantages over Arrays

There are some disadvantages of using linked lists when compared to arrays though.

- Only sequential access: As the data is linked together through nodes, any node can only be accessed by the node linking to it, hence it is not possible to randomly access any node. One has to go through the links searching for the element required.
- Memory Usage of each node: The nodes that hold the data need extra memory to hold the pointer to the next node. Each element hence takes slightly more memory than an array.

Creating a Linked List

For the linked list to be created, we need to define a node first depending on the type of linked list we want to create. Each type of list has specific properties and its own merits regarding the list operations.

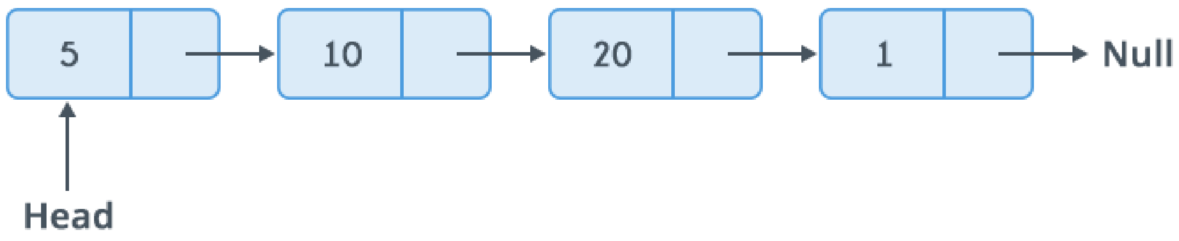
Types of Linked Lists

A linked list is designed depending on its use. The 3 most common types of a linked list are:

1. Singly Linked List
2. Doubly Linked List
3. Circular Linked List

Singly Linked List

This is the most common type of linked list, where each node has one pointer to the next node in the sequence. This means that the list can only be traversed from the beginning to the end in one direction. To access the last element, it is always required to traverse the whole list to the end.



The last node always points to NULL in a singly-linked list. This specifies that the list has ended with no more nodes to traverse to. Every time a loop traverses through the array, it checks for this NULL condition to know if the end of the linked list is there.

Implementation

Defining the Node

The Node contains 2 parts, one that is the data itself and the other which references the next node in the sequence. For simplicity, we will consider a Node where the data is a single integer. The data is not just limited to one value, one can define any number of pieces of information to be stored in each node.

In C, the node is defined as a structure . This type of a structure is called a self-referential structure where one member of the structure points to the structure of its kind.

```
struct Node
{
   int data;
   struct Node *next;
} *head = NULL;
```

A new Node is created first with the desired variable name. We will call this newNode for now.

```
struct Node *newNode;
```

The data stored in this Node can be accessed by using the arrow character (->) to the data member of the structure.

```
newNode->data
```

Similarly, the link to the next Node in the list can be accessed by using the arrow character to the *next member of the structure.

```
 newNode->next
```

About the head pointer node

The head node is used to point to the first node in a linked list. This is used to keep track of the list beginning and helps during the traversing operations.

Operations in a Linked List

The few basic operations in a linked list including adding, deleting and modifying.

Creating An Empty List

An empty list has to be created before performing any other operations. The head variable is created and assigned NULL. This will be used as the starting point to our linked list.

Adding To The End Of The List

New data can be added to the end of the linked list by creating a new Node with the data to be used, traversing to the end of the list and then appending this data to the end.

```
void insertAtEnd(int value)
{
   struct Node *newNode;
   newNode = (struct Node*)malloc(sizeof(struct Node));
   newNode->data = value;
   newNode->next = NULL;
   if(head == NULL)
      head = newNode;
   else
   {
     struct Node *temp = head;
       while(temp->next != NULL)
   temp = temp->next;
       temp->next = newNode;
   }
   printf("\nNode inserted successfully at end\n");
}
```

Adding To The Beginning Of The List

New data can be added to the beginning of the linked list by creating a new Node with the data to be used, replacing the head pointer to the new node and modifying the connections.

```
void insertAtBeginning(int value)
{
   struct Node *newNode;
   newNode = (struct Node*)malloc(sizeof(struct Node));
   newNode->data = value;
   if(head == NULL)
   {
      newNode->next = NULL;
      head = newNode;
    }
   else
    {
      newNode->next = head;
      head = newNode;
    }
    printf("\nNode inserted successfully at beginning\n");
}
```

Adding To A Specific Position Of The List

New data can be added at any position in the list by traversing to that position using a loop, creating a new Node and then manipulating the links to insert it at that position.

```
void insertPosition(int value, int pos)
{
```

```
int i = 0;
struct Node *newNode;
newNode = (struct Node*)malloc(sizeof(struct Node));
newNode->data = value;
if(head == NULL)
{
   newNode->next = NULL;
    head = newNode;
}
 else {
       struct Node *temp = head;
       for (i = 0; i < pos - 1; i++) {
           temp = temp-> next;
       }
       newNode->next = temp->next;
       temp->next = newNode;
 }

 printf("\nNode inserted successfully\n");
}
```

Deletion From The End Of The List

New data can be added to the end of the linked list by creating a new Node with the data to be used, traversing to the end of the list and then appending this data to the end.

```
void removeEnd()
{
 if(head == NULL)
 {
    printf("\nList is Empty\n");
 }
 else
```

```
  {
      struct Node *temp1 = head,*temp2;
       if(head->next == NULL)
         head = NULL;
       else
       {
          while(temp1->next != NULL)
          {
              temp2 = temp1;
              temp1 = temp1->next;
          }
          temp2->next = NULL;
       }
       free(temp1);
       printf("\nNode deleted at the end\n\n");
   }
}
```

Deletion From The Beginning Of The List

New data can be added to the beginning of the linked list by creating a new Node with the data to be used, replacing the head pointer to the new node and replacing the connections.

```
void removeBeginning()
{
   if(head == NULL)
  printf("\n\nList is Empty");
   else
   {
      struct Node *temp = head;
      if(head->next == NULL)
      {
         head = NULL;
```

```
        free(temp);
      }
      else
      {
         head = temp->next;
         free(temp);
         printf("\nNode deleted at the beginning\n\n");
      }
   }
}
```

Deletion From A Specific Position Of The List

New data can be deleted at any position in the list by traversing to that position using a loop, deleting the required Node and then manipulating the links to make the list continuous.

```
void removePosition(int pos)
{
   int i,flag = 1;

   if (head==NULL)
        printf("List is empty");
   else {
      struct Node *temp1 = head, *temp2;
      if (pos == 1) {
            head = temp1->next;
            free(temp1);
            printf("\nNode deleted\n\n");
       }
       else {
          for (i = 0; i < pos - 1; i++)
          {
             if (temp1 -> next != NULL) {
                    temp2 = temp1;
                    temp1 = temp1 -> next;
                }
                else {
                    flag = 0;
                    break;
                }
            }
            if (flag) {
                temp2 -> next = temp1 -> next;
                free(temp1);
                printf("\nNode deleted\n\n");
            }
            else {
                printf("Position exceeds number of elements in linked list. Please try again");
            }
        }
    }
}
```

Searching In A Linked List

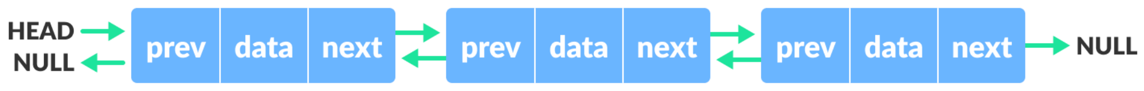
An element can be searched in a list by going through each element and checking it against the required element. As the element Nodes can only be accessed linearly, only a linear search can be performed in the case.

This is one of the disadvantages of a linked list regarding random access of elements.

```
void search(int key)
{
    while (head != NULL)
    {
        if (head->data == key)
        {
            printf("The key is found in the list\n");
            return;
        }
         head = head->next;
    }
     printf("The Key is not found in the list\n");
}
```

Doubly Linked List

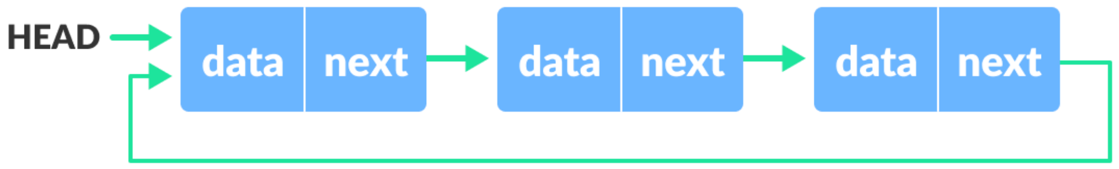
A doubly linked list has 2 pointers, one pointing to the next node and one to the previous node. This allows for moving in any direction while traversing the list, which may be useful in certain situations.



The implementation and details are here: Link to Doubly linked list

Circular Linked List

A circular linked list is like a regular one except for the last element of the list pointing to the first. This has the advantage of allowing to go back back to the first element while traversing a list without starting over.



Complexity of operations:

It is not possible to have a constant access time in linked list operations. The data required may be at the other end of the list and the worst case may be to traverse the whole list to get it.

Access

The elements of a linked list are only accessible in a sequential manner. Hence to access any element, we have to iterate through each node one by one until we reach the required element. The time complexity is hence O(n).

Insertion

Insertion in a linked list involves only manipulating the pointers of the previous node and the new node, provided we know the location where the node is to be inserted. Thus, the insertion of an element is O(1).

Deletion

Similar to deletion, deletion in a linked list involves only manipulating the pointers of the previous node and freeing the new node, provided we know the location where the node is to be deleted. Thus, the deletion of an element is O(1).

Applications of a Linked List

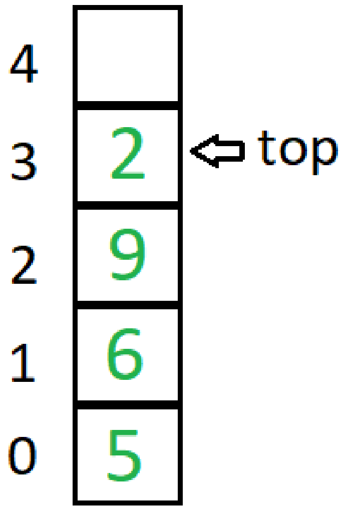
This is one of the most used data structures.

1. It is used to implement other data structures such as stacks, queues and non-linear ones like trees and graphs.
2. It has uses in hash chaining for the implementation in open chaining.

3. Polynomials can be represented and manipulated by using linked lists.
4. It can be used to perform operations on long integers.

Stacks

A stack is a linear data structure that store data in an order known as the Last In First Out (LIFO) order. This property is helpful in certain programming cases where the data needs to be ordered.



Stack

Stacks can be visualised like a stack of plates on a table. Only the top plate is accessible by the user at any given instant. The other plates are hidden and are not accessible by the user. The last plate that is kept on the stack is retrieved first.

Operations in a Stack

The two primary operations in a stack are the push and the pop operations:

Push Operation

This is used to add (or push) an element to the stack. The element always gets added to the top of the current stack items.

Pop Operation

This is used to remove (or pop) an element from the stack. The element always gets popped off from the top of the stack.

Peek Operation

The peek operation is used to return the first element of the stack without removing the element. It is a variation of the pop operation.

Overflow and Underflow Conditions

A stack may have a limited space depending on the implementation. We must implement check conditions to see if we are not adding or deleting elements more than it can maximum support.

The underflow condition checks if there exists any item before popping from the stack. An empty one cannot be popped further.

```
if (top == -1) {
    // underflow condition
}
```

The overflow condition checks if the stack is full (or more memory is available) before pushing any element. This prevents any error if more space cannot be allocated for the next item.

```
if (top == sizeOfStack) {
    // overflow condition
}
```

About The Top Pointer

To efficiently add or remove data, a special pointer is used which keeps track of the last element inserted in the structure. This pointer updates continuously and keeps a check on the overflow and underflow conditions.

Creating A Stack

A stack can be created using both an array or through a linked list. For simplicity, we will create one with an array.

1. First, we create a one-dimensional array with fixed size (`int stack[SIZE]`). The SIZE value could be defined using a preprocessor.
2. Define a integer variable `top` and initialize with '-1' (`int top = -1`).

```
#define SIZE 10

int stack[SIZE];
int top = -1;
```

Pushing To The Stack

Steps

1. Check whether stack is FULL. (`top == SIZE-1`)
2. If it is FULL, then terminate the function and throw an error.

3. If it is NOT FULL, then increment top value by one (top++) and set stack[top] to value (stack[top] = value).

```
void push(int value) {
  if(top == SIZE-1)
    printf("\nOverflow. Stack is Full");
  else{
    top++;
    stack[top] = value;
    printf("\nInsertion was successful");
  }
}
```

Popping From The Stack

Steps

1. Check whether stack is EMPTY. (top == -1)
2. If it is EMPTY, then terminate the function and throw an error.
3. If it is NOT EMPTY, then delete stack[top] and decrement top value by one (top--).

```
void pop() {
  if(top == -1)
    printf("\nUnderflow. Stack is empty");
  else{
    printf("\nDeleted : %d", stack[top]);
    top--;
  }
}
```

Accessing The Top Element (Peeking)

Steps

1. Check whether stack is EMPTY (top == -1).
2. If it is EMPTY, then terminate the function and throw an error.
3. If it is NOT EMPTY, then return stack[top].

```
void peek() {
  if(top == -1)
  {
     printf("\n The stack is empty");
     break;
  }
  else
     printf("%d", stack[top]);
}
```

Stack Complexity

Access

An element in a stack can only be accessed by continuously removing the front element until the required element is found. This means that the time complexity is O(n).

Search

Similar to accessing an element, searching an element will involve continuously popping an element until the required element is found. The time complexity is hence O(n).

Insertion

Inserting an element is only possible at the top of the stack. There is no interaction needed with the rest of the elements. It is hence an O(1) operation.

Deletion

Similar to insertion, deleting an element is only possible from the top of the stack. There is no interaction needed with the rest of the elements. It is hence an O(1) operation.

Space Required

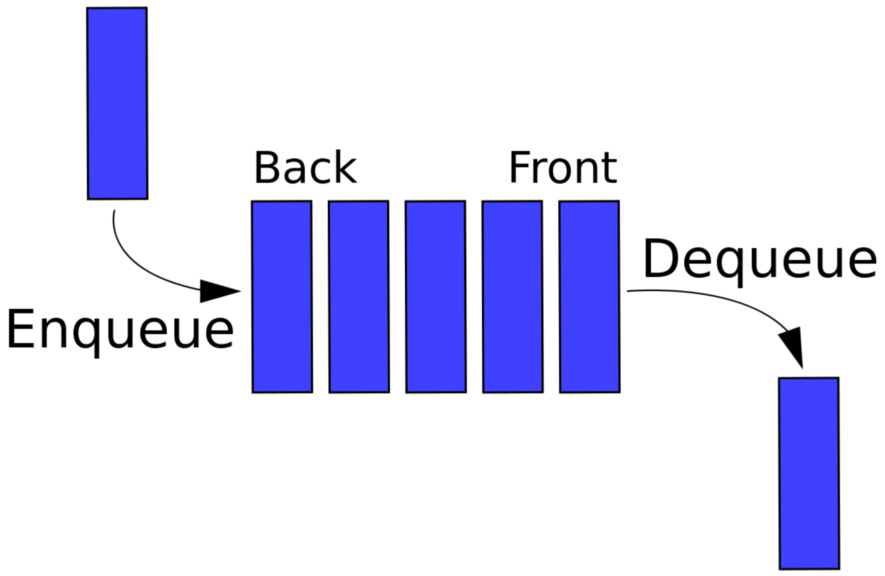
A stack only takes the space used to store the elements of the data type specified. This means that for storing n elements the space required is O(n).

Applications of Stacks in Programming

1. UNDO functionality in text editors: Every change in the document is added to stack and upon a UNDO request, the last change is referred by popping it.
2. Parentheses checker: The ordered manner of the stack could be used for checking the proper closing of parentheses. Every opening parentheses is pushed on to the stack and for every correct closing parentheses, it is popped off. Irregularities can then be detected if they mismatch.
3. Expression parsing: Using stacks can help evaluate expressions faster using postfix or prefix notation.

Queues

A queue is a linear data structure that stores data in an order known as the First In First Out order. This property is helpful in certain programming cases where the data needs to be ordered.



Queues can be visualised like a real-life queue of people. A person may join the queue from the end and may leave tit from the front. The first person to enter leaves first.

A ticket counter can be an example where the people standing in the queue get their tickets one by one and leave the queue.

Operations in a Queue

The two primary operations in a queue are the enqueue and the dequeue operation:

Enqueue Operation

The Enqueue is used to add an element to the queue. The element always gets added to the end of the current queue items.

Dequeue Operation

The Dequeue is used to remove an element from the queue. The element always gets removed from the front of the queue.

The Front And Rear Pointer

To efficiently add or remove data from the queue, two special pointers are used which keep track of the first and last element in the queue. These pointers update continuously and keep a check on the overflow and underflow conditions.

The front pointer always points to the position where an element would be dequeued next. The rear pointer always points to the position where an element would be enqueued next.

Overflow and Underflow Conditions

A queue may have a limited space depending on the implementation. We must implement check conditions to see if we are not adding or deleting elements more than it can maximum support.

The underflow condition checks if there exists any item before popping from the queue. An empty one cannot be dequeued further.

```
if(front == rear)
   // underflow condition
```

The overflow condition checks if the queue is full (or more memory is available) before enqueueing any element. This prevents any error if more space cannot be allocated for the next item.

```
if(rear == SIZE-1)
    // overflow condition
```

Creating A Queue

A queue can be created using both an array or through a linked list. For simplicity, we will create a queue with an array.

1. Create a one dimensional array with the above defined SIZE. (int queue[SIZE])
2. Define two integer variables front and rear and initialize both with '-1'. (int front = -1, rear = -1)

```
#define SIZE 10

int queue[SIZE];
int front = -1, rear = -1;
```

Enqueue Operation

1. Check whether the queue is FULL (rear == SIZE - 1).
2. If it is FULL, then display an error and terminate the function.
3. If it is NOT FULL, then increment the rear value by one (rear++) and set queue[rear] = value.

```
void enQueue(int value) {
  if(rear == SIZE-1)
    printf("\nOverflow. Queue is Full.");
  else{
    if(front == -1)
  front = 0;
    rear++;
    queue[rear] = value;
    printf("\nInsertion was successful");
   }
}
```

Dequeue Operation

1. Check whether the queue is EMPTY. (front == rear)
2. If it is EMPTY, then display an error and terminate the function.
3. If it is NOT EMPTY, then increment the front value by one (front++). Then display the queue[front] as the deleted element.
4. Then check whether both front and rear are equal (front == rear), if it TRUE, then set both front and rear to '-1' (front = rear = -1).

```
void deQueue() {
  if(front == rear)
     printf("\nUnderflow. Queue is Empty.");
  else{
     printf("\nDeleted item is: %d", queue[front]);
     front++;
     if(front == rear)
  front = rear = -1;
  }
}
```

Variations of a Queue

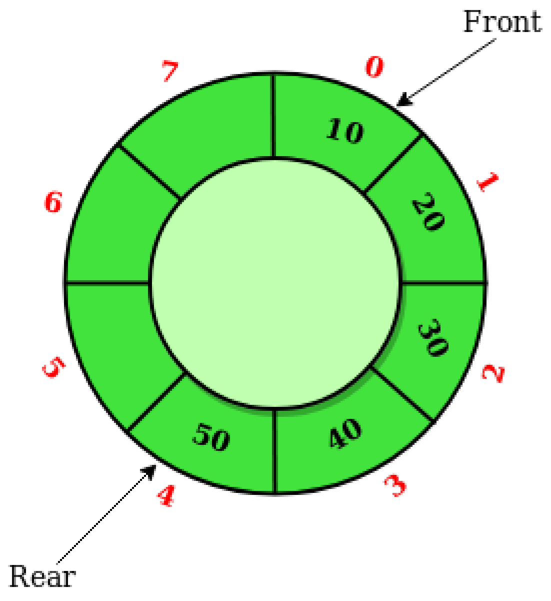
A queue can have some variations which make it useful in certain situations:

Double-Ended queue (Deque)

In a standard queue, insertion can only be done from the back and deletion only from the front. A double-ended queue allows for insertion and deletion from both ends.

Circular Queue (Circular Buffer)

A circular queue uses a single, fixed-size buffer as if it were connected end-to-end like a circle.



This is an efficient implementation for a queue that has fixed maximum size. There is no shifting involved and the whole queue can be used for storing all the elements.

Priority Queue

A priority queue assigns a priority to each element in the queue. This priority determines which elements are to be deleted and processed first. There can be different criteria's for the priority queue to assign priorities.

An element with the highest priority gets processed first. If there exist two elements with the same priority, then the order of which the element was inserted is considered.

Queue Complexity

Access

An arbitrary element in a queue can only be accessed by continuously shifting the front element. The time complexity is hence O(n).

Search

Similarly, searching an element will involve continuously shifting the front element off the queue until the required element is found. The time complexity is hence O(n).

Insertion

Inserting an element is only possible at the rear. There is no interaction needed with the rest of the elements. It is hence an O(1) operation.

Deletion

Similar to insertion, deleting an element is only possible from the front of the queue. There is no interaction needed with the rest of the elements. It is hence an O(1) operation.

Space Required

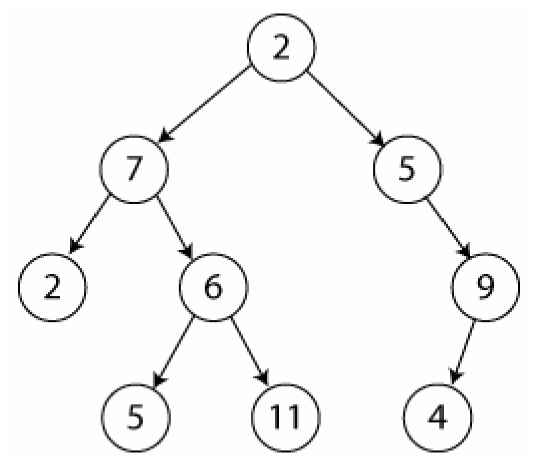
A queue only takes the space used to store the elements of the data type specified. This means that for storing n elements, the space required is O(n).

Applications of Queues in Programming

1. CPU Scheduling: Various CPU scheduling algorithms make use of this data structure to implement multiprocessing.
2. Synchronization during data transfer: Asynchronous data transfers use a queue to keep track of the data. This includes pipes and IO buffers.

Trees

A tree is a data structure that simulates a hierarchical tree, with a root value and the children as the subtrees, represented by a set of linked nodes. The children of each node could be accessed by traversing the tree until the specified value is reached.



This is a non-linear data structure unlike the other types of data structures like arrays, stacks and queues.

Basic Terminology

Before exploring trees, we need to learn of the basic terminologies associated with them:

Root: The first node in a tree is called as Root Node. Every tree must have one Root Node.

Parent Node: The node which is a predecessor of any node is called a Parent Node, that is, the node which has a branch from it to any other node is called as the Parent node.

Child Node: The node which is descendant of any node is called as Child Node. Any parent node can have any number of child nodes. All the nodes except root are child nodes.

Siblings: Nodes which belong to the same Parent are called as Siblings.

Leaf Node: In a tree data structure, the node which does not have a child is called a Leaf Node. They are also known as External Nodes or Terminal Nodes.

Internal Nodes: The node which has at least one child is called an Internal Node.

External Nodes: The node which has no child is called an External Node.

Degree: The total number of children of a node is called a Degree of that Node. The highest degree of a node among all the nodes in a tree is called the Degree of the tree.

Level: In a tree, each step from top to bottom is called a Level.

Height: The total number of edges from the leaf node to a particular node in the longest path is called as Height of that Node.

Depth: The total number of edges from the root node to a particular node is called the Depth of that Node.

Path: The sequence of Nodes and Edges from one node to another node is called a Path.

Types of A Tree

There are multiple types of trees with their various properties:

1. General trees
2. Binary trees
3. Binary Search trees
4. M-way trees
5. AVL trees

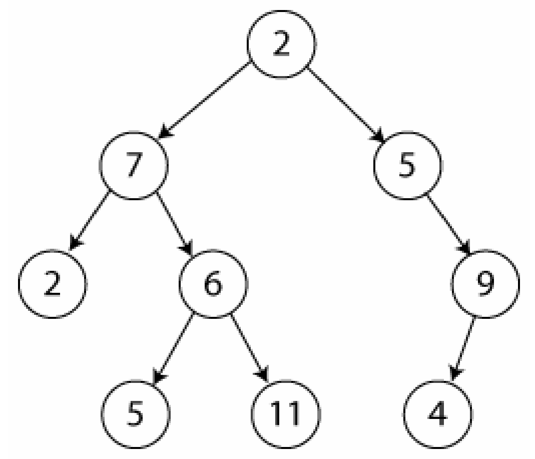
General Tree

A general tree is a tree where each node may have zero or more children. The other types of trees are special cases of general trees.

Mathematically it can be defined as a finite non-empty set of elements. One of these elements is called the *root* and the remaining elements when partitioned into trees, are called the subtrees of the root.

Binary Tree

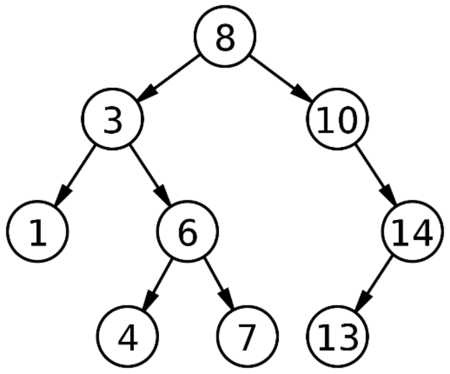
In a normal tree, each node can have any number of children. A Binary tree is a special case of general tree in which every node can have a maximum of two children. One is known as the left child and the other as right child.



Binary Search Trees

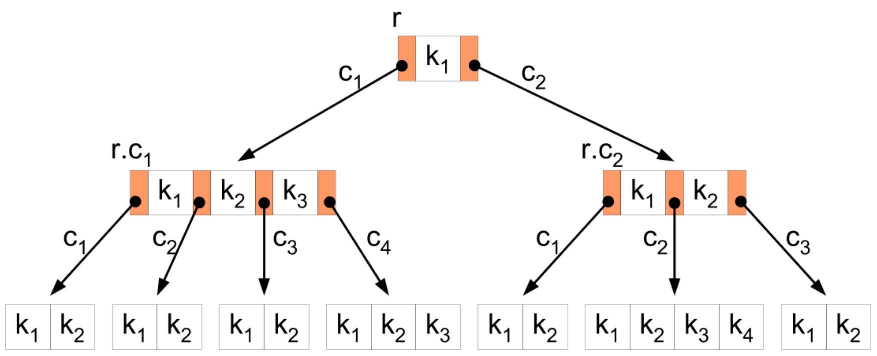
A Binary Search Tree is a binary tree that additionally satisfies the binary search property. This tree is used to decreases the number of comparisons to be made in the tree to find an element, like a regular binary search.

The binary search property states that the key in each node must be greater than or equal to any key stored in the left sub-tree, and less than or equal to any key stored in the right sub-tree.



Multiway Trees

A multiway tree can have more than one value per node. They are written as m-way trees where m means the order of the tree. A multiway tree can have m-1 values per node and m children. It is not necessary that every node has m-1 values or m children.



The 2 most used variants of multiway trees are:

1. B-Trees

A B-tree is a specialized M-way tree that is widely used for disk access. A B tree of order m can have a maximum of m–1 keys and m pointers to its sub-trees.

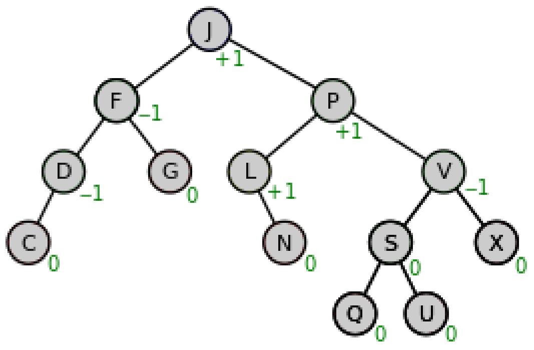
It was developed in the year 1972 by Bayer and McCreight. A B-tree is designed to store sorted data and allows search, insertion, and deletion operations to be performed in logarithmic running time.

2. B+ Trees

A B+ tree can be viewed as a B-tree in which each node contains only keys (not key-value pairs), and it stores all the records at the leaf level of the tree instead.

AVL Trees

An AVL tree is a self-balancing binary search tree . A binary tree is said to be balanced, if the difference between the heights of left and right subtrees of every node in the tree is either -1, 0 or +1.

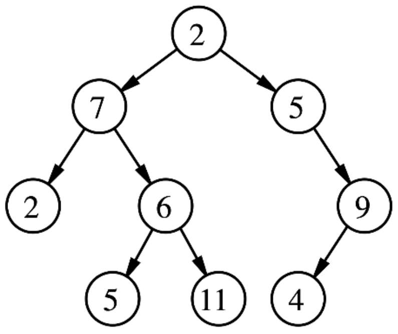


Applications of Trees in Programming

- File System structure: The directories and subdirectories of a file system are efficiently be represented by a tree structure.
- DOM structure: HTML pages are rendered using a DOM structure which contains all the tags used in the page. This is a tree-like structure.
- Router algorithms: Router algorithms construct a tree of the locations across the network to determine the route that data packets must follow to reach their destination efficiently.

Heaps

A heap is a complete binary tree that satisfies the heap property. There are two types of heaps, the max heap and the min heap.



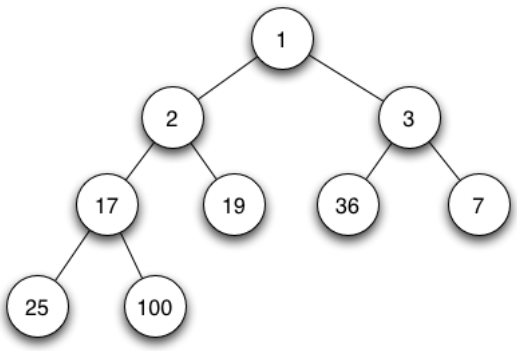
The Heap Property

The heap property says that is the value of Parent is either greater than or equal to (in a max heap) or less than or equal to (in a min heap) the value of the Child.

A heap is described in memory using linear arrays in a sequential manner.

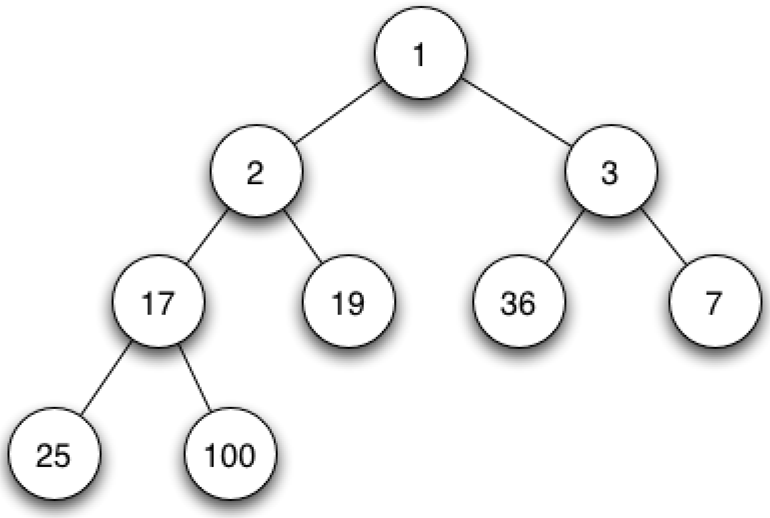
Max Heap

In a max heap, the key present at the root is the largest in the heap and all the values below this are less than this value.



Min Heap

In a min heap, the key present at the root is the smallest in the heap and all the values below this are greater than this value.



Heap Operations

The following operations can be performed on a heap data structure:

Insertion

A new element is always inserted at the last child of the original heap. The new element may now violate the heap property that a heap must satisfy.

Therefore, an operation known as reheapify upward is performed on the heap. The aim of the reheapify operation is to compare the new value inserted with its parent's value.

If the value is greater (in a max heap) or smaller (in a a min heap) it is swapped with its parent. The process is then continued from the parent node recursively until the heap property is satisfied or the root node is hit.

Deletion

An element is always deleted from the root of the heap. But deleting an element will leave a hole in the heap, which disturbs the requirement that the heap must be a complete binary tree. To fill this hole, the last node in the heap is swapped to this place. This causes the heap to violate the heap property.

Similar to inserting an element, an operation known as reheapify downward is performed on the heap. The value of the root node is first replaced with the largest (or smallest) value amongst its children. Then the down heap property is repeated from this child again recursively until we hit the leaf node.

Summing up the algorithm:

1. Replace the root node's value with the last node's value.
2. Delete the last node.
3. Sink down the new root node's value so that the heap again satisfies the heap property.

Finding Maximum/Minimum

Finding the node which has maximum or minimum value is easy due to the heap property and is one of the advantages of using a heap.

Since all the elements below it are smaller (or larger in a min-heap), it will be always the root node. This can be accessed in constant time.

Application in Programming

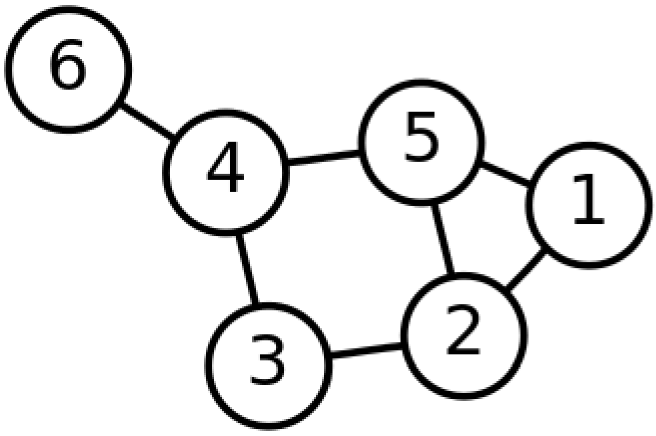
1. Heapsort: This is one of the best in-place sorting methods with no quadratic worst-case scenarios. This is because the minimum or maximum element is always the root of the heap.

2. Implementing priority queues: As the highest (or lowest) priority element is always stored at the root of the heap, they could be accessed quickly.
3. Selection algorithms: A heap allows access to the min or max element in constant time, and other selections (such as median or kth-element) can be done in sub-linear time on data that is in a heap.
4. Graph algorithms: By using heaps as internal traversal data structures, run times can be reduced by polynomial order.

Graphs

A graph data structure is used to represent relations between pairs of objects .

It consists of nodes (known as vertices) that are connected through links (known as edges). The relationship between the nodes can be used to model the relation between the objects in the graph. This is what makes graphs important in the real world.



It can be viewed as a generalization of the tree data structure as any kind of relationship can exist between the nodes of a tree, instead of the purely parent-child relationship of a tree.

Mathematically, Graph G is an ordered set (V, E) where V(G) represents the set of elements, called vertices, and E(G) represents the edges between these vertices.
A graph can be classified into 2 types:

1. Undirected Graphs

An undirected graph does not have any directed associated with its edges. This means that any edge could be traversed in both ways.

Mathematically, an edge is represented by an unordered pair [u, v] and can be traversed from u to v or vice-versa.

2. Directed Graphs

A directed graph has a direction associated with its edges. This means that any edge could be traversed only in the way of the direction.

Mathematically, an edge is represented by an ordered pair [u, v] and can only be traversed from u to v.

Basic Terminology In A Graph

Vertex: An individual data element of a graph is called Vertex.

Edge: An edge is a connecting link between two vertices. An Edge is also known as Arc.

Mixed Graph: A graph with undirected and directed edges is said to be a mixed graph.

Origin: If an edge is directed, its first endpoint is said to be the origin of it.

Destination: If an edge is directed, its first endpoint is said to be the origin of it and the other endpoint is said to be the destination of the edge.

Adjacency: Two node or vertices are adjacent if they are connected through an edge.

Path: The Path represents a sequence of edges between the two vertices.

Degree: The total number of edges connected to a vertex is said to be the degree of that vertex.

In-Degree: In-degree of a vertex is the number of edges which are coming into the vertex.

Out-Degree: Out-degree of a vertex is the number of edges which are going out from the vertex.

Minimum Spanning Tree (MST): A minimum spanning tree (MST) is a subset of the edges of a connected, edge-weighted (un)directed graph that connects all the vertices, without any cycles and with the minimum possible total edge weight.

Simple Graph: A graph is said to be simple if there are no parallel and self-loop edges.

Directed acyclic graph (DAG): A directed acyclic graph (DAG) is a graph that is directed and without cycles connecting the other edges. This means that it is impossible to traverse the entire graph starting at one edge.

Weighted Graph: A weighted graph is a graph in which a number (known as the weight) is assigned to each edge. Such weights might represent for example costs, lengths or capacities, depending on the problem.

Complete Graph: A complete graph is a graph in which each pair of vertices is joined by an edge. A complete graph contains all possible edges.

Connected Graph: A connected graph is an undirected graph in which every unordered pair of vertices in the graph is connected. Otherwise, it is called a disconnected graph.

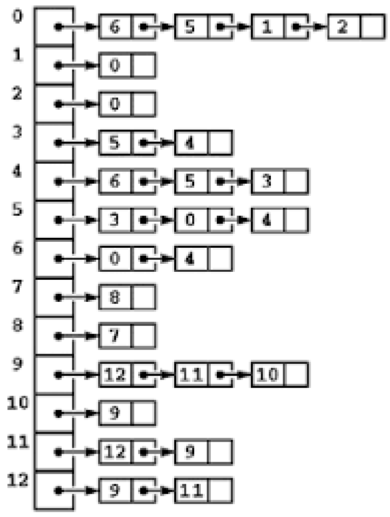
Representation Of A Graph

A binary graph data structure can be represented using two methods:

Adjacency List Representation

In this representation, every vertex of the graph contains a linked list of its neighboring vertices and edges.

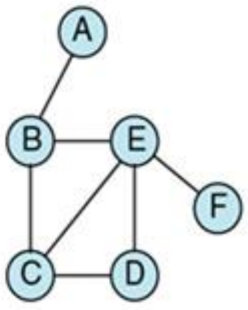
An array of lists is used where the size of the array is equal to the number of vertices. Each of the elements in the arrays contains a linked list of all the vertices adjacent to the list.



Adjacency Matrix Representation

In this representation, the graph can be represented using a matrix of size total number of vertices by the total number of vertices. Here, rows and columns both represent vertices. This matrix is filled with either 1 or 0.

Here, 1 represents there is an edge from row vertex to column vertex and 0 represents there is no edge from row vertex to a column vertex.



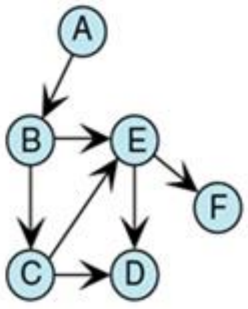
A
B
C
D
E
F

Vertex vector

	A	B	C	D	E	F
A	0	1	0	0	0	0
B	1	0	1	0	1	0
C	0	1	0	1	1	0
D	0	0	1	0	1	0
E	0	1	1	1	0	1
F	0	0	0	0	1	0

Adjacency matrix

(a) Adjacency matrix for non-directed graph



A
B
C
D
E
F

Vertex vector

	A	B	C	D	E	F
A	0	1	0	0	0	0
B	0	0	1	0	1	0
C	0	0	0	1	1	0
D	0	0	0	0	0	0
E	0	0	0	1	0	1
F	0	0	0	0	0	0

Adjacency matrix

(a) Adjacency matrix for directed graph

To represent the weights for weighted graphs, the weight of edge (u, v) is simply stored as the entry in row u and column v of the adjacency matrix.

Adjacency matrix representation requires O(V^2) memory locations irrespective of the number of edges in the graph.

Graph Traversal Algorithms

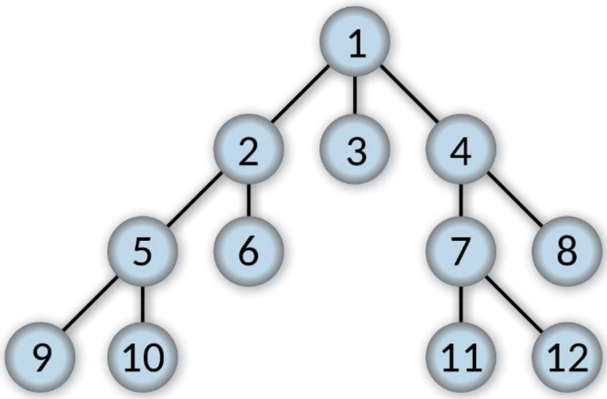
There are two standard graph traversal algorithms:

1. Breadth First Search (BFS)
2. Depth First Search (DFS)

Breadth First Search (BFS)

Breadth-first search begins at the root node of the graph and explores all its neighbouring nodes. For each of these nodes, the algorithm again explores its neighbouring nodes. This is continued until the specified element is found or all the nodes are exhausted.

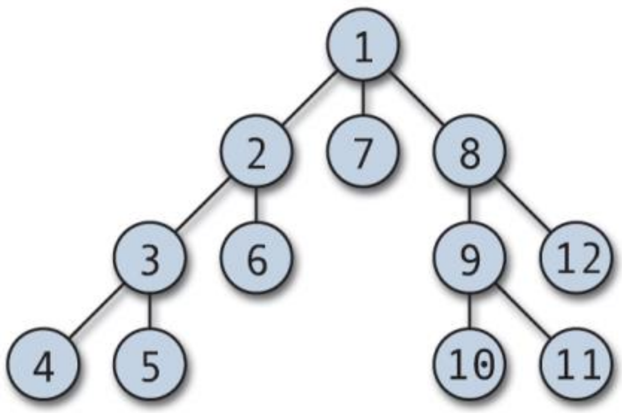
A queue is used as an auxiliary data structure to keep track of the neighboring nodes.



Depth First Search (DFS)

Depth-first search starts on a node and explores nodes going deeper and deeper until the specified node is found, or until a node with no children is found. If a node is found with no children, the algorithm backtracks and returns to the most recent node that has not been explored. This process continues until all the nodes have been traversed.

A stack is used as an auxiliary data structure to keep track of traversed nodes to help it backtrack when required.

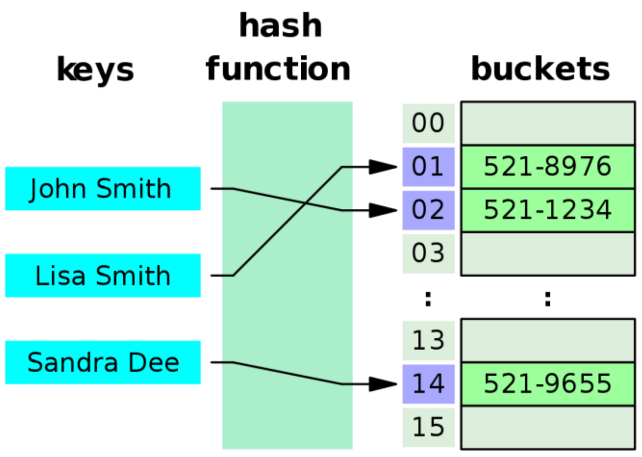


Applications of Graphs in Programming

- Family trees: can be mapped where the member nodes have an edge from parent to each of their children.
- Transportation networks: in which nodes are airports, intersections, ports, etc. The edges can be airline flights, one-way roads, shipping routes, etc.

Hash Tables

A hash table is a data structure where data is stored in an associative manner. The data is mapped to array positions by a hash function that generates a unique value from each key.



The value stored in a hash table can be searched in O(1) time, by using the same hash function which generates an address from the key. The process of mapping the keys to appropriate locations (or indices) in a hash table is called hashing.

Advantages of Hashing

The main advantage of hash tables over other data structures is speed . The access time of an element is on average O(1), therefore lookup could be performed very fast. Hash tables are particularly efficient when the maximum number of entries can be predicted in advance.

Hash Functions

A hash function is a mathematical formula which, when applied to a key, produces a value which can be used as an index for the key in the hash table.

The main aim of a hash function is that elements should be uniformly distributed. It produces a unique set of integers within some suitable range in order to reduce the number of collisions.

Properties of a Good Hash Function

Uniformity

A good hash function must map the keys as evenly as possible. This means that the probability of generating every hash value in the output range should roughly be the same. This also helps in reducing collisions.

Deterministic

A hash function must always generate the same hash value for a given input value.

Low Cost

The cost of executing a hash function must be small so that using the hashing technique becomes preferable over other traditional approaches.

Applications in Programming

1. Identification Databases: A hash function can make a unique signature from never changing data like our Date of Birth. This can then be used in combination with other variables to uniquely identify a person.
2. Search Engines: As the number of pages to be crawled is huge, a hash function can be used to determine if the page is unique or it had already been crawled before, without comparing the contents of the whole webpage.

Different Hash Functions

Division Method

This is the most simple method of hashing. Any integer, for example, x is divided by a number M and the remainder obtained is used as the hash.

Generally, M is chosen to be a prime number because a prime number increases the likelihood that the keys are mapped with uniformity in the output range of values.

This function could be represented as:

h(k) = k mod M

Multiplication Method

The Multiplication method has the following steps:

1. A constant is chosen which is between 0 and 1, say A.
2. The key k is multiplied by A.
3. The fractional part of kA is extracted.
4. The result of Step 3 is multiplied by the size of the hash table (m).

This can be represented as:

h(k) = fractional_part[m (kA mod 1)]

Mid-Square Method

The Mid-Square method is as follows:

1. The value of the key is squared. That is, k^2 is found.

2. The middle r digits of the result are extracted.
3. The result r is the hash obtained.

The algorithm works well because most or all digits of the key-value contribute to the resulting hash.

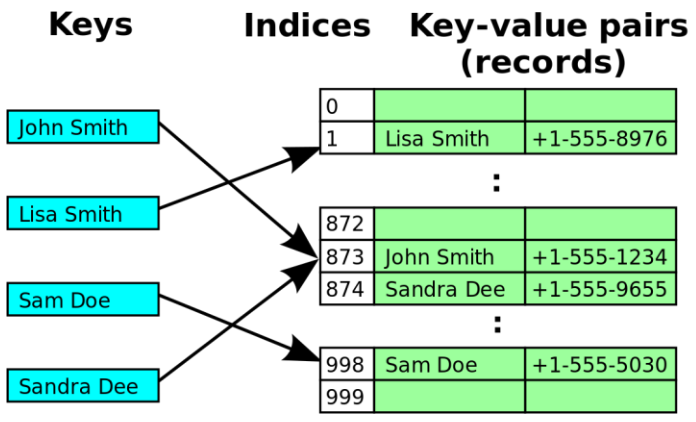
Collisions

Collisions occur when the hash function maps two different keys to the same location. Two records cannot be stored in the same location of a hash table normally.

The method used to solve the problem of collisions is called the collision resolution technique.

There are two popular collision resolution techniques:

Open Addressing



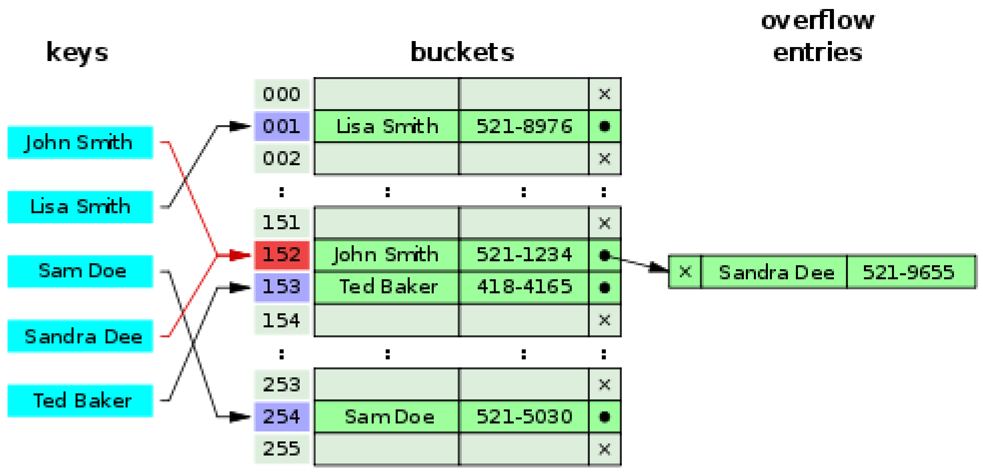
Once a collision takes place, open addressing (also known as closed hashing) computes new positions using a probe sequence and the next record is stored in that position. There are some well-known probe sequences:

1. Linear Probing: The interval between the probes is fixed to 1. This means that the very next available position in the table would be tried.

2. Quadratic Probing: The interval between the probes increases quadratically. This means that the next available position that would be tried would increase quadratically.
3. Double Hashing: The interval between probes is fixed for each record but the hash is computed again by double hashing.

Chaining

Chaining is another solution to the problem of collisions.



In chaining, each location in a hash table stores a pointer to a linked list that contains all the key values that were hashed to that location. As new collisions occur, the linked list grows to accommodate those collisions forming a chain.

This effectively means that each location in the hash table is not limited to store one value. Searching for a value in a chained hash table is as simple as scanning a linked list for an entry with the given key.

Insertion operation appends the key to the end of the linked list pointed by the hashed location.

Deleting a key requires searching the list and removing the element.

This solution, however, presents a problem if the linked list becomes large enough that it takes O(n) time to search one position. This occurs if the hash table is too small and has to accommodate many values.