Intro TO DSA

1. What is the significance of data structures in computer science? Discuss its importance in various fields.

Data structures are fundamental concepts in computer science that are used to organize and store data in a way that enables efficient access, modification, and retrieval of information. They are essential to the development of efficient algorithms and software systems that process large volumes of data.

The importance of data structures in various fields of computer science are discussed below:

1. Programming: In programming, data structures are essential tools for creating efficient algorithms and software systems. The choice of data structure used can greatly affect the performance and efficiency of the program. Some common data structures used in programming include arrays, lists, stacks, queues, trees, and graphs.
2. Database Systems: In database systems, data structures are used to organize and store data in an efficient and meaningful way. Data structures such as indexes, B-trees, hash tables, and heaps are commonly used to manage data in databases.
3. Artificial Intelligence and Machine Learning: In artificial intelligence and machine learning, data structures are used to store and process large datasets. Data structures such as matrices, tensors, and graphs are commonly used to represent and manipulate data in AI and ML applications.
4. Networking: In networking, data structures are used to represent and manage network topology, routing information, and other network-related data. Data structures such as trees, graphs, and hash tables are commonly used to manage network-related data.
5. Operating Systems: In operating systems, data structures are used to manage system resources such as memory, processes, and files. Data structures such as queues, stacks, and linked lists are commonly used in operating systems to manage system resources efficiently.

In summary, data structures are critical concepts in computer science that enable the efficient organization and manipulation of data. They are used in a wide range of fields such as programming, database systems, artificial intelligence, machine learning, networking, and operating systems. A good understanding of data structures is essential for developing efficient algorithms and software systems in these fields.

1. Explain the different types of operations that can be performed on data structures.

Data structures support various types of operations that can be performed on the data stored in them. The operations supported by a data structure depend on the type of data structure and its implementation. Some common types of operations that can be performed on data structures are as follows:

1. Traversal: Traversal refers to the process of visiting each element of a data structure in a systematic manner. Traversal operations are commonly used in data structures such as arrays, lists, trees, and graphs. There are several ways to traverse a data structure, such as in-order, pre-order, and post-order traversal in trees.
2. Insertion: Insertion refers to the process of adding a new element to a data structure. Insertion operations are commonly used in data structures such as arrays, lists, and trees. The time complexity of insertion varies depending on the type of data structure used.
3. Deletion: Deletion refers to the process of removing an element from a data structure. Deletion operations are commonly used in data structures such as arrays, lists, and trees. The time complexity of deletion also varies depending on the type of data structure used.
4. Searching: Searching refers to the process of finding the location or value of a particular element in a data structure. Searching operations are commonly used in data structures such as arrays, lists, trees, and hash tables. There are several algorithms used for searching, such as binary search, linear search, and depth-first search.
5. Sorting: Sorting refers to the process of arranging the elements of a data structure in a particular order. Sorting operations are commonly used in data structures such as arrays and lists. There are several algorithms used for sorting, such as bubble sort, insertion sort, selection sort, and quicksort.
6. Merging: Merging refers to the process of combining two or more data structures into a single data structure. Merging operations are commonly used in data structures such as arrays and lists.
7. Splitting: Splitting refers to the process of dividing a data structure into two or more smaller data structures. Splitting operations are commonly used in data structures such as arrays and lists.

In summary, data structures support various types of operations, including traversal, insertion, deletion, searching, sorting, merging, and splitting. These operations are essential to efficiently process and manipulate data stored in data structures.

1. What are abstract data types, and why are they important in data structure design?

Abstract data types (ADTs) are a way of describing a collection of related data and the operations that can be performed on that data. An ADT specifies the interface for a data structure, but it does not specify how the data structure is implemented.

ADTs are important in data structure design because they provide a high-level abstraction of a data structure that is independent of any specific implementation. By separating the interface from the implementation, ADTs make it possible to change the underlying implementation of a data structure without affecting the interface or the clients that use the data structure.

ADTs provide a number of benefits in data structure design, including:

1. Encapsulation: ADTs encapsulate the details of the data structure implementation, which makes it easier to reason about and maintain the code. By hiding the implementation details, the ADT provides a clear separation of concerns between the interface and the implementation.
2. Abstraction: ADTs provide a high-level abstraction of a data structure that is independent of the specific implementation. This allows the designer to focus on the interface and the operations that the data structure should support, rather than the details of the implementation.
3. Modularity: ADTs make it possible to modularize the code by separating the interface from the implementation. This makes it easier to reuse code and to modify the implementation without affecting the clients that use the data structure.
4. Portability: ADTs provide a portable interface to a data structure that is independent of the specific platform or programming language used. This makes it easier to write code that can be ported to different platforms or programming languages.

In summary, abstract data types are important in data structure design because they provide a high-level abstraction of a data structure that is independent of the specific implementation. This separation of concerns makes it easier to reason about and maintain the code, and it provides a number of benefits such as encapsulation, abstraction, modularity, and portability.

1. What is an array list, and how is it different from a regular array? Explain the operations that can be performed on an array list.

An array list is a dynamic array implementation that can grow or shrink in size as needed. It is a data structure that provides the benefits of both arrays and linked lists. In an array list, elements are stored in a contiguous block of memory, just like in a regular array. However, an array list can dynamically resize itself by allocating a new block of memory and copying the elements from the old block to the new one.

The main difference between an array list and a regular array is that an array list is dynamic and can resize itself, while a regular array has a fixed size and cannot be resized once it is created. This means that an array list can grow or shrink in size as needed, while a regular array requires that the size be fixed at the time of creation.

Some common operations that can be performed on an array list are:

1. Insertion: Elements can be inserted into an array list at a specific index. If the index is beyond the end of the array list, the array list will automatically resize to accommodate the new element.
2. Deletion: Elements can be deleted from an array list at a specific index. If the index is beyond the end of the array list, an error may be raised.
3. Access: Elements can be accessed in an array list by their index, just like in a regular array.
4. Search: Elements can be searched for in an array list using linear search or binary search algorithms, just like in a regular array.
5. Resizing: The size of an array list can be changed dynamically as needed, unlike a regular array.
6. Traversal: Elements can be traversed in an array list using a loop or iterator.

In summary, an array list is a dynamic array implementation that provides the benefits of both arrays and linked lists. It can grow or shrink in size as needed, and supports various operations such as insertion, deletion, access, search, resizing, and traversal.

1. What is the difference between dynamic and static data structures? Discuss the advantages and disadvantages of each type.
2. The main difference between dynamic and static data structures is that dynamic data structures can change size at runtime, while static data structures have a fixed size that is determined at compile time.
3. Dynamic data structures, such as linked lists and array lists, have the advantage of flexibility. They can grow or shrink in size as needed, which means they can accommodate changing amounts of data. This makes them suitable for applications where the amount of data to be stored is unknown or varies over time. Dynamic data structures also have efficient insertion and deletion operations since elements can be added or removed without having to move other elements.
4. However, dynamic data structures can also have some disadvantages. They require more memory overhead than static data structures since they may need to allocate and deallocate memory frequently. This can result in performance overhead, especially if the data structure needs to be resized frequently. Dynamic data structures may also require more complex implementation and can be harder to debug.
5. On the other hand, static data structures, such as arrays and stacks, have the advantage of being more memory-efficient since they allocate a fixed amount of memory upfront. This means that they are more suitable for applications where the amount of data is known and does not change frequently. Static data structures also have simpler implementation and are easier to debug.
6. However, static data structures also have some disadvantages. They cannot grow or shrink in size at runtime, which means they may not be suitable for applications where the amount of data can vary over time. Insertion and deletion operations can be inefficient since they require moving other elements. Additionally, static data structures may lead to memory wastage if the allocated memory is not fully utilized.
7. In summary, dynamic data structures provide flexibility and efficient insertion and deletion operations, but may have memory overhead and complex implementation. Static data structures are more memory-efficient and have simpler implementation, but may not be suitable for applications where the amount of data varies frequently and may have inefficient insertion and deletion operations. The choice between dynamic and static data structures depends on the specific requirements of the application.
8. What are pointers, and how are they used in data structures? Provide examples.

Pointers are variables that store memory addresses of other variables or data structures. They are widely used in data structures to manipulate data in memory, as they allow efficient access and modification of data without having to copy the entire data structure.

class Node

{

private:

int Data;

Node \*Next;

public:

Node(int x)

{

Data=x;

Next=NULL;

}

No

1. How are structures used in data structures? Explain with an example.

In data structures, structures are used to represent complex data types that can be composed of multiple fields or attributes. A structure is a collection of related data items that can be of different types, such as integers, floating-point numbers, characters, or even other structures. Structures allow us to create custom data types that can be used to represent real-world objects or concepts.

#include <string>

class Person {

public:

// Constructor

Person(std::string n, int a, std::string addr) {

name = n;

age = a;

address = addr;

}

// Getter methods

std::string getName() { return name; }

int getAge() { return age; }

std::string getAddress() { return address; }

// Setter methods

void setName(std::string n) { name = n; }

void setAge(int a) { age = a; }

void setAddress(std::string addr) { address = addr; }

private:

std::string name;

int age;

std::string address;

};

1. Compare and contrast arrays and linked lists in terms of their storage allocation and operations that can be performed on them.

Arrays and linked lists are both fundamental data structures used in computer science, but they differ in their storage allocation and the operations that can be performed on them.

Storage Allocation:

* Arrays use contiguous memory allocation to store elements in a block of memory, where each element is accessed using an index. This means that arrays have a fixed size and require continuous block of memory.
* Linked lists use dynamic memory allocation to store elements, where each element (node) is allocated individually and connected through pointers. Each node contains a data element and a pointer to the next node in the list, which allows for flexible size.

Operations:

* Arrays offer constant time access to any element, as elements are stored contiguously in memory and accessed by their index. They also offer constant time for appending and deleting elements from the end of the array.
* Linked lists offer constant time insertion and deletion of elements at any position in the list, as only the affected nodes need to be updated. However, accessing an element in a linked list requires traversing the list, which takes linear time in the worst case.

In summary, arrays are a better choice when constant-time access to elements and fixed-size storage is important. Linked lists, on the other hand, are a better choice when dynamic size, constant-time insertion and deletion, and flexible memory allocation are important. Here is a comparison table summarizing the differences between arrays and linked lists:

| **Feature** | **Array** | **Linked List** |
| --- | --- | --- |
| Storage Allocation | Contiguous Memory | Dynamic Memory |
| Accessing Elements | Constant Time | Linear Time |
| Inserting/Deleting Elements | Linear Time | Constant Time |
| Fixed/Dynamic Size | Fixed | Dynamic |
| Memory Allocation | Static | Dynamic |

Overall, the choice between arrays and linked lists depends on the specific use case and the requirements of the program.

1. Explain the concept of memory allocation in dynamic data structures, and provide examples of how it can be managed.

n dynamic data structures, memory allocation refers to the process of reserving and managing memory space for the data structure at run-time. Unlike static data structures, which have a fixed size determined at compile-time, dynamic data structures can grow or shrink in size as needed during the execution of the program.

There are two main types of memory allocation: stack allocation and heap allocation.

* Stack allocation: In stack allocation, memory is allocated in a contiguous block of memory known as the stack. When a function is called, its local variables are pushed onto the top of the stack, and when the function returns, the variables are popped off the stack. This type of memory allocation is fast and efficient, but limited in size, and the size must be known at compile-time.
* Heap allocation: In heap allocation, memory is allocated on the heap, which is a large pool of memory available to the program. This type of memory allocation allows for dynamic allocation of memory at run-time, and can be used for large data structures with variable sizes. However, it is slower and less efficient than stack allocation, and requires explicit memory management to avoid memory leaks and other issues.

1. Discuss the role of data structures in algorithm design and analysis.

Data structures play a crucial role in algorithm design and analysis. In computer science, algorithms are sets of instructions that solve a specific computational problem, and data structures are used to organize and store data for efficient processing by algorithms.

Here are some ways in which data structures are used in algorithm design and analysis:

1. Efficiency: Data structures can significantly impact the efficiency of an algorithm. Choosing the right data structure can improve the running time and memory usage of an algorithm. For example, using a hash table for fast lookups or a priority queue for efficient sorting.
2. Problem modeling: Data structures are used to model real-world problems in computer science. For example, a graph data structure can be used to model a transportation network, and a tree data structure can be used to model a hierarchical structure.
3. Algorithm development: Data structures can inspire the development of new algorithms. For example, the data structure known as the suffix tree led to the development of an efficient algorithm for string searching.
4. Algorithm analysis: The analysis of algorithms involves determining their running time and memory usage. Data structures are used to store and manipulate data during the execution of an algorithm, and their use can impact the analysis of the algorithm.

Linked List

1. What is the difference between a singly linked list, a doubly linked list, and a circular linked list?
2. Singly linked list: In a singly linked list, each node contains a reference to the next node in the sequence, but not to the previous node. The last node in the sequence has a null reference. Singly linked lists can be traversed only in one direction, from the head node to the tail node. Singly linked lists are simple to implement and efficient for certain operations, such as adding and removing elements from the head of the list, but less efficient for other operations, such as searching and removing elements from the middle of the list.
3. Doubly linked list: In a doubly linked list, each node contains references to both the next node and the previous node in the sequence. The first node in the sequence has a null reference to the previous node, and the last node has a null reference to the next node. Doubly linked lists can be traversed in both directions, from the head node to the tail node and vice versa. Doubly linked lists are more complex to implement than singly linked lists, but they provide faster access to elements in the middle of the list and more flexibility for certain operations, such as reversing the list or iterating in reverse order.
4. Circular linked list: In a circular linked list, the last node in the sequence has a reference to the first node, creating a circular structure. Circular linked lists can be singly or doubly linked. They are useful for certain applications, such as implementing a round-robin scheduling algorithm or modeling a circular buffer.

2.How is a doubly linked list implemented in memory and what are its advantages and disadvantages over a singly linked list?

A doubly linked list is implemented in memory using nodes that contain three fields: a data element, a reference to the previous node, and a reference to the next node. The first node in the list is called the head node, and its previous node reference is null. The last node in the list is called the tail node, and its next node reference is null.

#include<iostream>

using namespace std;

class node{

public:

int data;

node\*next,\*prev;

node(int val){

data=val;

next=NULL;prev=NULL;

}

};

void insertAtHead(node\* &head,int val){

node\*n=new node(val);

// node\*temp=head;

n->next=head;

if(head!=NULL){

head->prev=n;

}

head=n;

}

void insertAtTail(node\* &head,int val){

if(head==NULL){

insertAtHead(head,val);

return;

}

node\* n=new node(val);

node\*temp=head;

while(temp->next!=NULL){

temp=temp->next;

}

temp->next=n;

n->prev=temp;

}

void print(node\*head){

node\*temp=head;

while(temp!=NULL){

cout<<temp->data<<"->";

temp=temp->next;

}

cout<<"NULL"<<endl;

}

void deleteHead(node\* &head){

node\*todelete=head;

head=head->next;

head->prev=NULL;

delete todelete;

}

void deletion(node\* &head,int val){

node\*temp=head;

if(temp->data==val){

deleteHead(head);

return;

}

while(temp!=NULL && temp->data!=val){

temp=temp->next;

}

temp->prev->next=temp->next;

if(temp->next!=NULL){

temp->next->prev=temp->prev;

}

delete temp;

}

int main(){

node\*head=NULL;

insertAtTail(head,1);

insertAtTail(head,2);

insertAtTail(head,3);

insertAtTail(head,4);

insertAtTail(head,5);

insertAtTail(head,6);

print(head);

insertAtHead(head,7);

print(head);

deletion(head,6);

print(head);

// deleteHead(head);

// print(head);

return 0;

}Advantages of doubly linked list over singly linked list:

1. Bi-directional traversal: A doubly linked list allows traversal in both directions, from head to tail and tail to head. This enables faster traversal to the end of the list and makes certain operations, such as reverse traversal, easier to perform.
2. Efficient removal of nodes: In a doubly linked list, a node can be removed from the list without the need to traverse the entire list to find its predecessor. This can lead to faster removal of nodes from the list.

Disadvantages of doubly linked list over singly linked list:

1. Increased memory usage: A doubly linked list requires more memory to store the extra reference to the previous node in each node. This can be a disadvantage if memory usage is a concern.
2. More complex implementation: A doubly linked list is more complex to implement than a singly linked list, as it requires maintaining references to both the previous and next nodes in each node. This can make the implementation more error-prone and difficult to maintain.
3. How do you insert a new node at the beginning of a doubly linked list?
4. void insertAtHead(node\* &head,int val){
5. node\*n=new node(val);
6. // node\*temp=head;
7. n->next=head;
8. if(head!=NULL){
9. head->prev=n;
10. }
11. head=n;
12. }How do you insert a new node at the end of a doubly linked list?

void insertAtTail(node\* &head,int val){

if(head==NULL){

insertAtHead(head,val);

return;

}

node\* n=new node(val);

node\*temp=head;

while(temp->next!=NULL){

temp=temp->next;

}

temp->next=n;

n->prev=temp;

}

void print(node\*head){

node\*temp=head;

while(temp!=NULL){

cout<<temp->data<<"->";

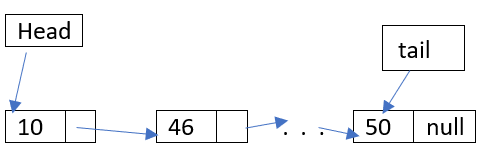
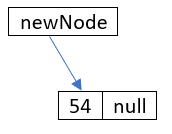
temp=temp->next;

}

cout<<"NULL"<<endl;

}

5. Consider following lines of algo is applied to a singly linked list with head and tail node as shown in the figure. Re ***sketch*** the linked list after the given algorithm is ***executed*** for the list.

1. How do you delete a node from a doubly linked list?

void deleteHead(node\* &head){

node\*todelete=head;

head=head->next;

head->prev=NULL;

delete todelete;

}

void deletion(node\* &head,int val){

node\*temp=head;

if(temp->data==val){

deleteHead(head);

return;

}

while(temp!=NULL && temp->data!=val){

temp=temp->next;

}

temp->prev->next=temp->next;

if(temp->next!=NULL){

temp->next->prev=temp->prev;

}

delete temp;

}}

1. How do you traverse a doubly linked list in both forward and backward directions?

void deleteNodeAtPos(Node\*\* headRef, int pos) {

// base case: empty list

if (\*headRef == NULL) {

return;

}

// find the node at the given position

Node\* curNode = \*headRef;

for (int i = 1; i < pos && curNode != NULL; i++) {

curNode = curNode->next;

}

// if position is out of range, return

if (curNode == NULL) {

return;

}

// case 1: deleting the head node

if (\*headRef == curNode) {

\*headRef = curNode->next;

}

// case 2: deleting a node in the middle or at the end

if (curNode->next != NULL) {

curNode->next->prev = curNode->prev;

}

if (curNode->prev != NULL) {

curNode->prev->next = curNode->next;

}

// delete the node

delete curNode;

}

1. How is a circular linked list different from a regular linked list and what are its use cases?
2. A circular linked list is similar to a regular linked list, but the last node in the list points back to the first node instead of NULL, forming a circular chain. This means that a circular linked list can be traversed indefinitely, unlike a regular linked list which has a clear end.
3. One advantage of a circular linked list is that it can be useful for implementing data structures that need to have a cyclical nature. For example, a circular buffer or a circular queue can be implemented using a circular linked list. A circular linked list can also be used for representing a circular path or loop in a graph data structure.
4. Another advantage of a circular linked list is that it allows for constant-time insertion and deletion at both the beginning and end of the list, since there is no need to update NULL pointers. However, finding the end of the list requires special handling, since there is no clear end.

10. How do you insert a new node at the beginning of a circular linked list?

void insertBegin(int data) {

Node\* newNode = new Node(data);

if (this->head == NULL) {

this->head = newNode;

this->head->next = this->head;

} else {

newNode->next = this->head;

Node\* lastNode = getLastNode();

lastNode->next = newNode;

this->head = newNode;

}

}

1. How do you insert a new node at the end of a circular linked list?

// delete the last node of the list

void deleteEnd() {

if (this->head == NULL) {

return;

}

if (this->head->next == this->head) {

delete this->head;

this->head = NULL;

} else {

Node\* lastNode = getLastNode();

Node\* temp = this->head;

while (temp->next != lastNode) {

temp = temp->next;

}

temp->next = this->head;

delete lastNode;

}

}

1. How do you delete a node from a circular linked list?

void deleteNodeByPosition(int position) {

if (head == NULL) {

return;

}

Node\* temp = head;

if (position == 1) {

if (temp->next == head) {

head = NULL;

free(temp);

return;

}

head = temp->next;

temp->prev->next = head;

head->prev = temp->prev;

free(temp);

return;

}

int currentPosition = 1;

while (currentPosition != position && temp->next != head) {

temp = temp->next;

currentPosition++;

}

if (currentPosition != position) {

return;

}

temp->prev->next = temp->next;

temp->next->prev = temp->prev;

free(temp);

}

Stacks

* 1. How can you implement a stack using an array and what are the advantages and disadvantages of this approach?

#include<iostream>

using namespace std;

class Stack

{

private:

int \*arr;

int nextIndex;

int capacity;

public:

Stack()

{

capacity=4;

arr=new int[capacity];

nextIndex=0;

}

Stack(int cap)

{

capacity=cap;

arr=new int[capacity];

nextIndex = 0;

}

int size()

{

return nextIndex;

}

void push(int element){

if (nextIndex==capacity)

{

cout << "Stack is Full";

return;

}

else

{

arr[nextIndex]=element;

nextIndex++;

}

}

int top()

{

return arr[nextIndex];

}

};

int main()

{

}

Advantages:

* Array-based stacks have constant time complexity for all operations because the elements are stored in contiguous memory locations, making it easy to access and modify them.
* Array-based stacks are easy to implement and understand.

Disadvantages:

* The size of the array used to implement the stack must be fixed at the time of creation, which means that the stack can hold a limited number of elements.
* If the array becomes full, we need to create a new array of larger size and copy all the elements from the old array to the new array, which can be time-consuming and inefficient.
  1. How can you implement a stack using a linked list and what are the advantages and disadvantages of this approach?

#include<iostream>

using namespace std;

class Stack

{

class Node

{

public:

int data;

Node \*next;

Node(int d)

{

data=d;

next=NULL;

}

};

int stacksize;

Node \*head;

public:

Stack()

{

head=NULL;

stacksize=0;

}

bool isEmpty()

{

if(head==NULL)

return true;

return false;

}

void push(int ele)

{

Node \*temp=new Node(ele);

temp->next=head;

head=temp;

stacksize++;

}

void pop()

{

if (isEmpty())

{

cout << "Stack is Empty...";

return;

}

Node\* temp=head;

head=temp->next;

temp->next=NULL;

delete temp;

stacksize--;

}

int top()

{

if (isEmpty())

{

cout << "Stack is Empty...";

return -1;

}

return head->data;

}

int size()

{

return stacksize;

}

};

int main()

{

Stack s;

s.push(5);

s.push(56);

s.push(34);

s.push(34);

while (!s.isEmpty())

{

cout << s.top() << endl;

s.pop();

}

}

Advantages:

* The size of the stack can be dynamic and grow or shrink as needed.
* Insertion and deletion of elements can be done efficiently with constant time complexity O(1).
* Implementation is easy to understand and modify.

Disadvantages:

* Extra memory is required for storing the pointers to the next node.
* Random access is not possible, so it may not be suitable for certain algorithms that require direct access to the elements.
  1. How can you use two stacks to implement a queue and what are the advantages and disadvantages of this approach?

#include <stack>

template <typename T>

class Queue {

private:

std::stack<T> s1, s2;

public:

void enqueue(T val) {

s1.push(val);

}

T dequeue() {

if (s2.empty()) {

while (!s1.empty()) {

s2.push(s1.top());

s1.pop();

}

}

T val = s2.top();

s2.pop();

return val;

}

bool empty() {

return s1.empty() && s2.empty();

}

int size() {

return s1.size() + s2.size();

}

};

* 1. How can you use a stack to evaluate arithmetic expressions and what is the algorithm for this process?

Stacks can be used to evaluate arithmetic expressions by using the postfix notation (also known as Reverse Polish notation) instead of the usual infix notation. The algorithm for evaluating an arithmetic expression in postfix notation using a stack is as follows:

1. Create an empty stack.
2. Scan the expression from left to right.
3. If the current character is a number, push it onto the stack.
4. If the current character is an operator (+, -, \*, /, etc.), pop the two topmost values from the stack, apply the operator, and push the result back onto the stack.
5. When the expression has been completely scanned, the result is the value left on the stack.

For example, let's evaluate the expression "5 3 + 2 \*":

1. Create an empty stack.
2. Scan "5", which is a number, push it onto the stack.
3. Scan "3", which is a number, push it onto the stack.
4. Scan "+", which is an operator. Pop 3 and 5 from the stack, add them, and push the result (8) back onto the stack.
5. Scan "2", which is a number, push it onto the stack.
6. Scan "\*", which is an operator. Pop 2 and 8 from the stack, multiply them, and push the result (16) back onto the stack.
7. The expression has been completely scanned, and the result is 16, which is the value left on the stack.

The advantage of using a stack to evaluate arithmetic expressions in postfix notation is that it eliminates the need for parentheses and the order of operations is unambiguous. The disadvantage is that converting an expression from infix to postfix notation can be a complex process

1. Given a stack A containing unsorted integers, write a pseudocode which puts the elements in A in Decending order. You can only use an additional stack (say B) and some additional non-array variables to complete this task
2. Create an empty stack B.
3. While stack A is not empty:
   1. Pop the top element from stack A and store it in a variable temp.
   2. While stack B is not empty and the top element is less than temp:
      1. Pop the top element from stack B and push it onto stack A.
   3. Push temp onto stack B.
4. While stack B is not empty:
   1. Pop the top element from stack B and push it onto stack A.

6. How can you use a stack to implement a depth-first search algorithm in a graph and what are the steps involved in this process?

7.Evaluate the prefix expression "- + 1 \* 3 / 6 2 4 5" using a stack. Show the step-by-step evaluation process (i.e. contents of stack and result at each step) and provide the final result of the expression.

#include<iostream>

using namespace std;

class Stack

{

int \*arr;

int NextIndex;

int capacity;

public:

Stack()

{

capacity=4;

arr=new int[capacity];

NextIndex=0;

}

Stack(int cap)

{

capacity=cap;

arr=new int[capacity];

NextIndex=0;

}

int size()

{

return NextIndex;

}

bool isEmpty()

{

return NextIndex==0;

}

void push(int ele)

{

if(NextIndex==capacity)

{

capacity++;

int \*arr1=new int[capacity];

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

{

arr1[i]=arr[i];

}

arr1[NextIndex]=ele;

NextIndex++;

}

else

{

arr[NextIndex]=ele;

NextIndex++;

}}

void pop()

{

if(isEmpty())

{

cout << "Stack is Empty..." ;

return;

}

NextIndex--;

}

int top()

{

if (isEmpty())

{

cout << "Stack is Empty....";

return 0;

}

return arr[NextIndex-1];

}

};

int evaluatePreFix(string exp)

{

Stack s;

for (int i=exp.size();i>=0;i--)

{

if (isdigit(exp[i]))

{

int num=exp[i]-'0';

s.push(num);

}

else

{

int op1 = s.top();

s.pop();

int op2 = s.top();

s.pop();

if (exp[i] == '+')

s.push(op1 + op2);

else if (exp[i] == '-')

s.push(op1 - op2);

else if (exp[i] == '\*')

s.push(op1 \* op2);

else if (exp[i] == '/')

s.push(op1 / op2);

}

}

return s.top();

}

int main()

{

string s1="- + 1 \* 3 / 6 2 4 5" ;

int result =evaluatePreFix(s1);

cout << "Result = " << result ;

return 0;

}

NOTE: Read Prefix and Postfix in Detail

Queue

How can you implement a queue using an array and what are the advantages and disadvantages of this approach?

#include<iostream>

using namespace std;

#define n 20

class queues

{

private:

int \*arr;

int frontt;

int backk;

public:

queues()

{

arr=new int[n];

frontt=-1;

backk=-1;

}

void push(int x)

{

if (backk==n-1)

{

cout << "Queues Overflow..." ;

return;

}

backk++;

arr[backk]=x;

if (frontt==-1)

{

frontt++;

}

}

void pop()

{

if (frontt==-1 || frontt > backk)

{

cout << "QUEUE IS EMPTY...";

return;

}

frontt++;

}

int top()

{

if (frontt==-1 || frontt > backk)

{

cout << "QUEUE IS EMPTY...";

return -1;

}

return arr[frontt];

}

};

How can you implement a queue using a linked list and what are the advantages and disadvantages of this approach?

#include<iostream>

using namespace std;

class Queues

{

private:

class Node

{

public:

int data;

Node \*next;

Node (int val)

{

data=val;

next=NULL;

}

};

Node \*frontt;

Node \*backk;

public:

Queues ()

{

frontt=NULL;

backk=NULL;

}

void push(int val)

{

Node \*temp=new Node(val);

if (frontt==NULL)

{

backk=temp;

frontt=temp;

return;

}

backk->next=temp;

backk=temp;

}

void pop()

{

if (frontt==NULL)

{

cout << "Queue Underflow..";

return;

}

Node \*temp=frontt;

frontt=frontt->next;

delete temp;

}

int top()

{

if (frontt==NULL)

{

cout << "NO VALUE IN QUEUE..";

return -1;

}

return frontt->data;

}

};

int main()

{

}

How can you use two queues to implement a stack and what are the advantages and disadvantages of this approach?

#include<iostream>

#include<queue>

#include<stdexcept>

using namespace std;

class Stacks

{

queue <int> q1,q2;

int sizeofstack;

public:

Stacks()

{

sizeofstack=0;

}

void push(int value) {

q2.push(value);

while (!q1.empty()) {

q2.push(q1.front());

q1.pop();

}

swap(q1, q2);

sizeofstack++;

}

void pop() {

if (empty()) {

throw out\_of\_range("Stack is empty");

}

q1.pop();

sizeofstack--;

}

int top() const {

if (empty()) {

throw out\_of\_range("Stack is empty");

}

return q1.front();

}

};

int main()

{

}

How can you use a queue to implement a breadth-first search algorithm in a graph and what are the steps involved in this process?

How can we add and delete value from Double ended circular Queue

#include <iostream>

using namespace std;

class Deque {

private:

int\* arr;

int front;

int rear;

int capacity;

int count;

public:

Deque(int size) {

arr = new int[size];

front = -1;

rear = 0;

capacity = size;

count = 0;

}

~Deque() {

delete[] arr;

}

void insertFront(int value) {

if (isFull()) {

cout << "Deque is full." << endl;

return;

}

if (front == -1) {

front = 0;

rear = 0;

} else if (front == 0) {

front = capacity - 1;

} else {

front--;

}

arr[front] = value;

count++;

}

void insertRear(int value) {

if (isFull()) {

cout << "Deque is full." << endl;

return;

}

if (rear == capacity - 1) {

rear = 0;

} else {

rear++;

}

arr[rear] = value;

count++;

}

void deleteFront() {

if (isEmpty()) {

cout << "Deque is empty." << endl;

return;

}

if (front == rear) {

front = -1;

rear = 0;

} else if (front == capacity - 1) {

front = 0;

} else {

front++;

}

count--;

}

void deleteRear() {

if (isEmpty()) {

cout << "Deque is empty." << endl;

return;

}

if (front == rear) {

front = -1;

rear = 0;

} else if (rear == 0) {

rear = capacity - 1;

} else {

rear--;

}

count--;

}

int getFront() {

if (isEmpty()) {

cout << "Deque is empty." << endl;

return -1;

}

return arr[front];

}

int getRear() {

if (isEmpty()) {

cout << "Deque is empty." << endl;

return -1;

}

return arr[rear];

}

int size() {

return count;

}

bool isEmpty() {

return count == 0;

}

bool isFull() {

return count == capacity;

}

void display() {

if (isEmpty()) {

cout << "Deque is empty." << endl;

return;

}

int i = front;

while (i != rear) {

cout << arr[i] << " ";

i = (i + 1) % capacity;

}

cout << arr[rear] << endl;

}

};

int main() {

Deque q(5);

q.insertFront(1);

q.insertFront(2);

q.insertRear(3);

q.insertRear(4);

q.insertRear(5);

q.display(); // output: 2 1 3 4 5

q.deleteFront();

q.deleteRear();

q.display(); // output: 1 3 4

q.insertFront(6);

q.insertRear(7);

q.display(); // output: 6 1 3 4 7

return 0;

}

``

Tree

1.How can you implement a binary tree using an array and what are the advantages and disadvantages of this approach?

#include <iostream>

#include <cmath>

using namespace std;

class Binary\_Tree {

private:

int\* arr;

int capacity;

public:

Binary\_Tree(int size) {

arr = new int[size + 1];

capacity = size;

for (int i = 1; i <= capacity; i++) {

arr[i] = 0;

}

}

void insert(int value) {

if (arr[1] == 0) {

arr[1] = value;

return;

}

for (int i = 1; i <= capacity; i++) {

if (arr[i] == 0) {

arr[i] = value;

return;

}

}

}

void display() {

for (int i = 1; i <= capacity; i++) {

if (arr[i] != 0) {

cout << arr[i] << " ";

}

}

cout << endl;

}

int parent(int index) {

return arr[floor(index/2)];

}

int leftChild(int index) {

return arr[2\*index];

}

int rightChild(int index) {

return arr[2\*index+1];

}

};

int main() {

Binary\_Tree tree(10);

tree.insert(1);

tree.insert(2);

tree.insert(3);

tree.insert(4);

tree.insert(5);

tree.insert(6);

tree.insert(7);

tree.display(); // output: 1 2 3 4 5 6 7

cout << "Parent of index 4: " << tree.parent(4) << endl; // output: 2

cout << "Left child of index 2: " << tree.leftChild(2) << endl; // output: 4

cout << "Right child of index 2: " << tree.rightChild(2) << endl; // output: 5

return 0;

}

2.How can you implement a binary tree using a linked list and what are the advantages and disadvantages of this approach?

3. How can you traverse a binary tree in-order, pre-order, and post-order using both iterative and recursive approaches?

void inOrderTraversal(Node\* node) {

if (node == nullptr) {

return;

}

inOrderTraversal(node->left);

cout << node->data << " ";

inOrderTraversal(node->right);

}

void preOrderTraversal(Node\* node) {

if (node == nullptr) {

return;

}

cout << node->data << " ";

preOrderTraversal(node->left);

preOrderTraversal(node->right);

}

void postOrderTraversal(Node\* node) {

if (node == nullptr) {

return;

}

postOrderTraversal(node->left);

postOrderTraversal(node->right);

cout << node->data << " ";

}

4.How can you calculate the height, depth, and diameter of a binary tree and what is the algorithm for each of these operations?

int height(Node\* root) {

if(root == NULL)

return 0;

int leftHeight = height(root->left);

int rightHeight = height(root->right);

return max(leftHeight, rightHeight) + 1;

}

int depth(Node\* root) {

if(root == NULL)

return 0;

int leftDepth = depth(root->left);

int rightDepth = depth(root->right);

return max(leftDepth, rightDepth) + 1;

}

int diameter(Node\* root) {

if(root == NULL)

return 0;

int leftHeight = height(root->left);

int rightHeight = height(root->right);

int leftDiameter = diameter(root->left);

int rightDiameter = diameter(root->right);

return max(leftHeight + rightHeight + 1, max(leftDiameter, rightDiameter));

}

5. Given the following pre-order traversal of a binary tree, construct the binary tree and show the step-by-step process for generating the tree:

Pre-Order Traversal: 20, 10, 5, 12, 22, 21, 25

#include <iostream>

using namespace std;

struct Node {

int data;

Node\* left;

Node\* right;

};

Node\* constructTree(int arr[], int start, int end) {

if(start > end)

return NULL;

Node\* root = new Node;

root->data = arr[start];

root->left = root->right = NULL;

int i;

for(i = start; i <= end; i++) {

if(arr[i] > root->data)

break;

}

root->left = constructTree(arr, start + 1, i - 1);

root->right = constructTree(arr, i, end);

return root;

}

void inorderTraversal(Node\* root) {

if(root == NULL)

return;

inorderTraversal(root->left);

cout << root->data << " ";

inorderTraversal(root->right);

}

int main() {

int arr[] = {20, 10, 5, 12, 22, 21, 25};

int n = sizeof(arr) / sizeof(arr[0]);

Node\* root = constructTree(arr, 0, n - 1);

cout << "Inorder traversal of the constructed tree: ";

inorderTraversal(root);

return 0;

}

A. In the binary tree generated by the pre-order traversal in question 1, what is the height of the tree and what is the value of the root node?

B. Given a binary tree, how can you determine whether it is a binary search tree or not, using pre-order traversal?

C. How can you use pre-order traversal to convert a binary tree into a mirror image of itself?

void mirrorImage(Node\* root) {

if (root == NULL)

return;

mirrorImage(root->left);

mirrorImage(root->right);

Node\* temp = root->left;

root->left = root->right;

root->right = temp;

}

D. How can you use pre-order traversal to find the lowest common ancestor of two nodes in a binary tree?

Node\* lowestCommonAncestor(Node\* root, Node\* p, Node\* q) {

if (root == NULL || root == p || root == q)

return root;

Node\* leftLCA = lowestCommonAncestor(root->left, p, q);

Node\* rightLCA = lowestCommonAncestor(root->right, p, q);

if (leftLCA != NULL && rightLCA != NULL)

return root;

else if (leftLCA != NULL)

return leftLCA;

else

return rightLCA;

}

Note.Practice Question with Post Order