 **LAB MANUAL**

**SUBMITTED BY:**

AREEBA ASAD 21112011016

HEBA BINT E SAJID 21112011043

AMINA BIBI 21112011009

HUDA UMAR 21112011035

**SUBMITTED TO:**

SAMRA NASEER

1. **ARRAYS:**

**Array Representation:**

Arrays can be declared in various ways in different languages. For illustration, let's take C array declaration.



Arrays can be declared in various ways in different languages. For illustration, let's take C array declaration.

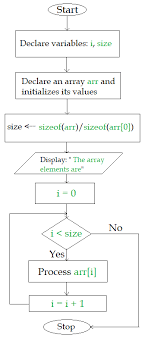


As per the above illustration, following are the important points to be considered.

* Index starts with 0.
* Array length is 10 which means it can store 10 elements.
* Each element can be accessed via its index. For example, we can fetch an element at index 6 as 9.

1. **TRAVERSING:**

This operation is used to print all the array elements one by one.



#include "stdafx.h"

#include<iostream>

using namespace std;

void traverse(int arr[] , int n)

{

cout << "Enter the data of array : " << endl;

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

{

cin >> arr[i];

}

cout << "The data of array is : " << endl;

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

{

cout << arr[i] << endl;;

}

}

void main()

{

int arr[50],n;

cout << "Enter the size of array" << endl;

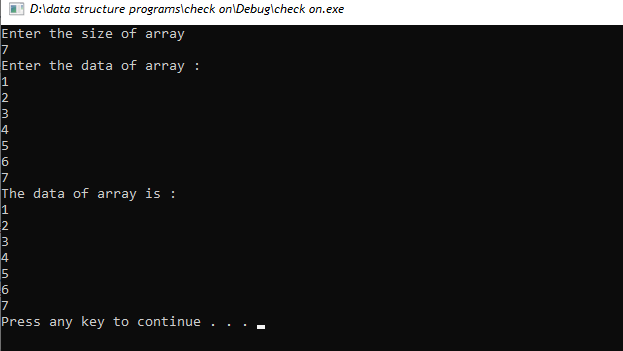
cin >> n;

traverse(arr, n);

system("pause");

return;

}



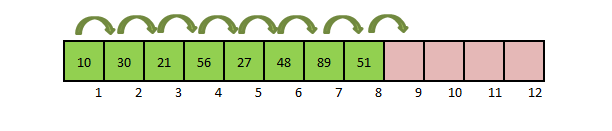
1. **INSERTION:**

Insert operation is to insert one or more data elements into an array. Based on the

requirement, a new element can be added at the beginning, end, or any given index of array.

* **INSERTION\_AT\_START:**

In this case we have to move all the elements one position backwards to make  a hole at the beginning of array. Though the insertion process is not difficult but freeing the first location for new element involves movement of all the existing elements. This is the worst case scenario in insertion in a linear array.



#include "stdafx.h"

#include<iostream>

using namespace std;

int main()

{

int arr[10], n, elem;

cout << "How many elements you want to insert = ";

cin >> n;

cout << "Enter data in an Array :: ";

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

{

cin >> arr[i];

}

cout << "\nEnter element you want to insert in First Position" << endl;

cin >> elem;

for (int i = n; i>0; i--)

{

arr[i] = arr[i - 1];

}

arr[0] = elem;

n = n + 1;

cout << "\n\nArray after Insertion :: \n";

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

{

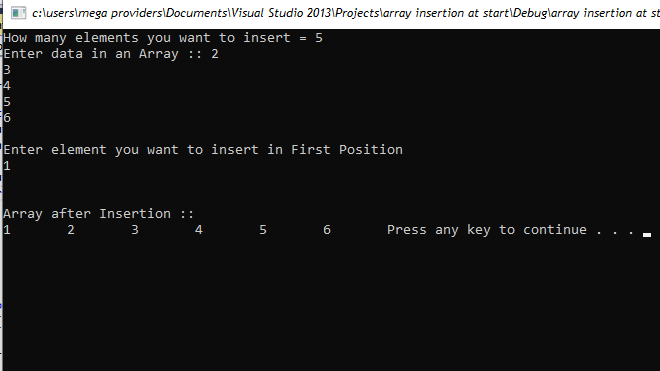
cout << arr[i] << "\t";

}

system("pause");

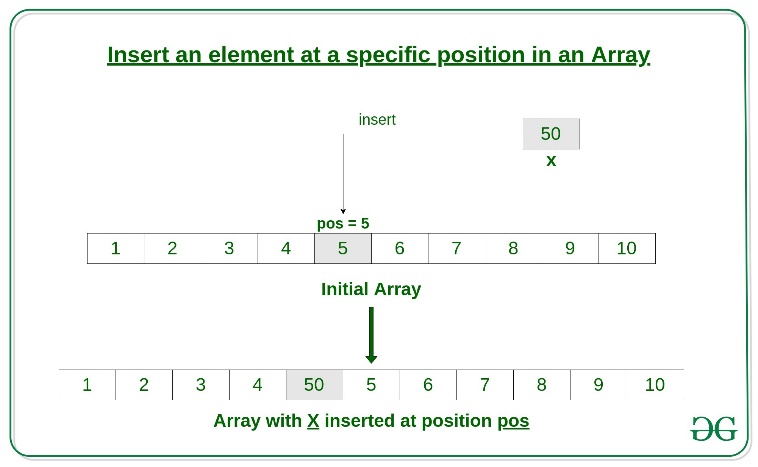
return 0;

}



* **INSERTION\_AT\_MIDDLE:**

The following figure shows how a value in inserted in a specific location.



#include "stdafx.h"

#include<iostream>

using namespace std;

void traverse(int arr[], int n)

{

cout << "The data of array is : " << endl;

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

{

cout << arr[i] << endl;;

}

}

void insertion(int arr[], int n)

{

int pos, element;

cout << "Enter the positon where u want insertion " << endl;

cin >> pos;

for (int i = n; i >= pos; i--)

{

arr[i] = arr[i - 1];

}

cout << "Enter new elemnt " << endl;

cin >> element;

arr[pos] = element;

n++;

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

{

cout << arr[i] << endl;

}

}

void main()

{

int arr[5] = { 4, 8, 9, 4, 7 }, n = 5;

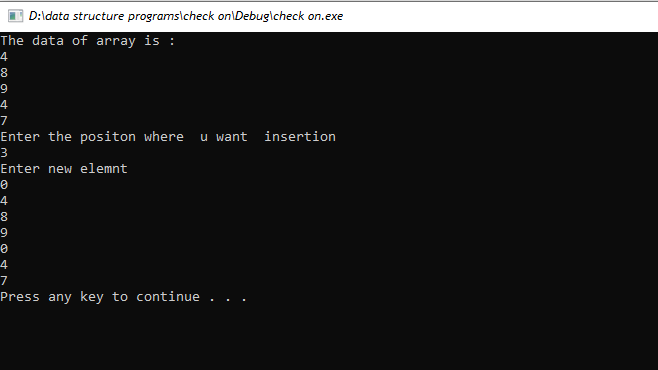
traverse(arr, n);

insertion(arr, n);

system("pause");

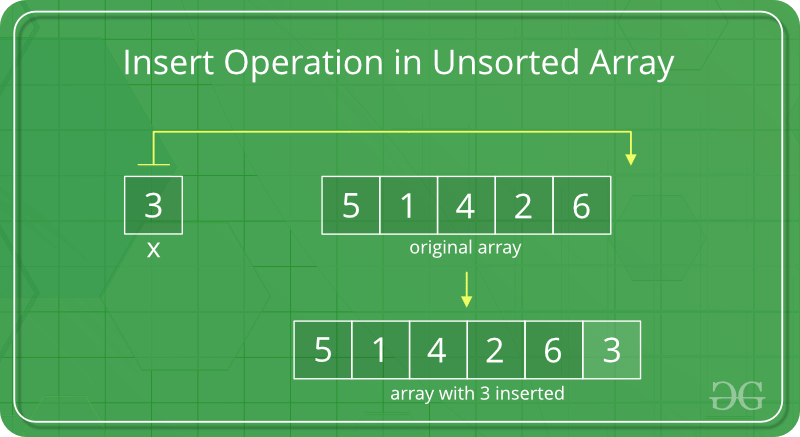
return;

}



* **INSERTION\_AT\_END:**

In an unsorted array, the insert operation is faster as compared to a sorted array because we don’t have to care about the position at which the element is to be placed. The following figure shows how a value in inserted in a specific location.



#include "stdafx.h"

#include<iostream>

using namespace std;

void traverse(int arr[], int n)

{

cout << "The data of array is : " << endl;

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

{

cout << arr[i] << endl;;

}

}

void insertion\_at\_end(int arr[], int n,int element)

{

arr[n] = element;

n++;

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

{

cout << arr[i] << endl;

}

}

void main()

{

int arr[5] = { 4, 8, 9, 4, 7 }, n = 5;

traverse(arr, n);

int element;

cout << "Enter new elemnt " << endl;

cin >> element;

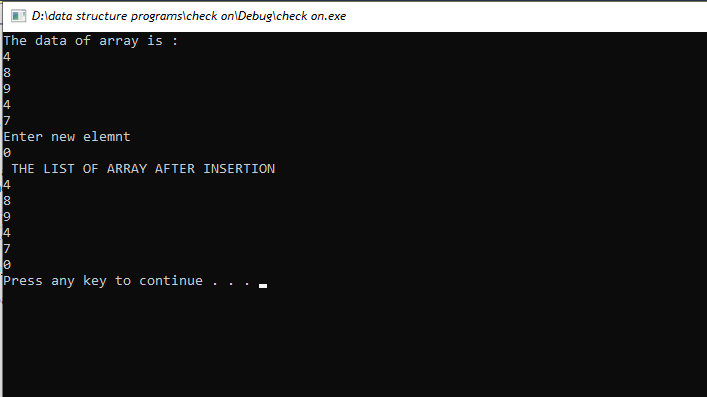
cout << " THE LIST OF ARRAY AFTER INSERTION " << endl;

insertion\_at\_end(arr, n,element);

system("pause");

return;

}



1. **UPDATION:**

Updating an element in an array means changing the value of an existing element in the array to a new value. This can be done by finding the index of the element to be updated and then updating the value at that index to the new value.

Code has been written below to explain the given concept:

#include "stdafx.h"

#include<iostream>

using namespace std;

void traverse(int arr[])

{

cout << "The data of array is : " << endl;

for (int i = 0; i < 5; i++)

{

cout << arr[i] << endl;;

}

}

void updation(int arr[])

{

int pos, value;

cout << "Enter the position where u want insertion: " << endl;

cin >> pos;

if (pos <= 5)

{

cout << "Enter the new value : " << endl;

cin >> value;

arr[pos - 1] = value;

cout << "Updated values are: " << endl;

for (int i = 0; i < 5; i++)

{

cout << arr[i] << endl;

}

}

else{

cout << "LIMIT OF ARRAY EXCEEDS" << endl;

}

}

void main()

{

int arr[5] = {4,7,2,8,9}, n;

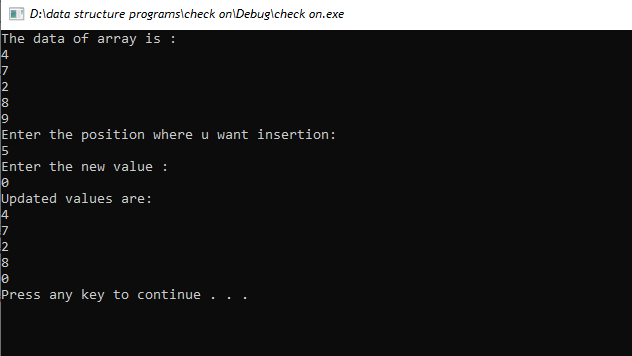
traverse(arr);

updation(arr);

system("pause");

return;

}



1. **DELETION:**

In the delete operation, the element to be deleted is searched using the [linear search](https://www.geeksforgeeks.org/linear-search/), and then the delete operation is performed followed by shifting the elements

* **DELETION\_AT\_START:**

#include "stdafx.h"

#include<iostream>

using namespace std;

void traverse(int arr[], int n)

{

cout << "The data of array is : " << endl;

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

{

cout << arr[i] << endl;;

}

}

void deletion\_at\_start(int arr[], int n)

{

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

{

if (arr[i] == arr[0])

{

for (int j = i; j < n - 1; j++)

{

arr[j] = arr[j + 1];

}

}

}

n--;

cout << "Upadated data after deletion is: " << endl;

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

{

cout << arr[i] << endl;

}

}

void main()

{

int arr[5] = { 4, 7, 9, 2, 1 }, n = 5;

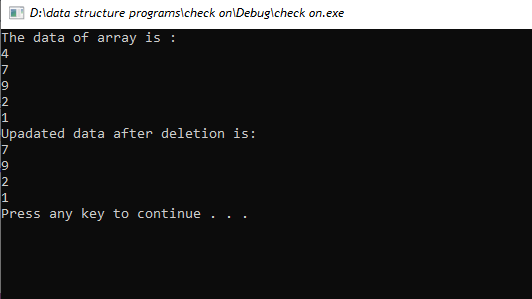
traverse(arr, n);

deletion\_at\_start(arr, n);

system("pause");

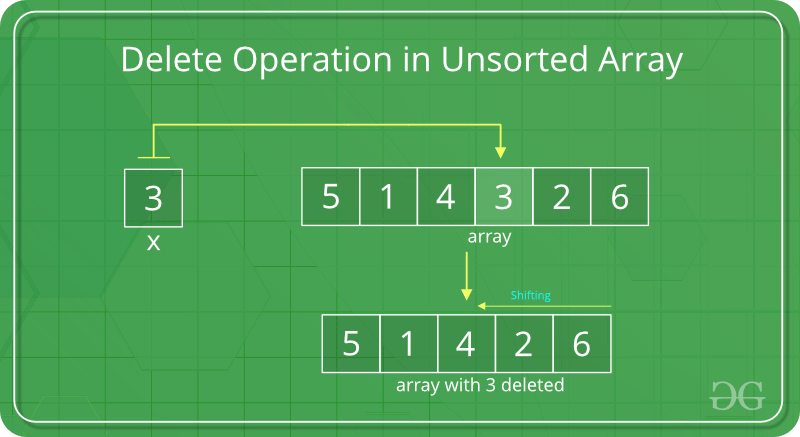
return;

}



* **DELETION\_AT\_MIDDLE:**

The following figure shows how a value in inserted in a specific location.



#include "stdafx.h"

#include<iostream>

using namespace std;

void traverse(int arr[], int n)

{

cout << "The data of array is : " << endl;

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

{

cout << arr[i] << endl;;

}

}

void deletion(int arr[], int n)

{

int val;

cout << "Enter the value u want to delete " << endl;

cin >> val;

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

{

if (arr[i] == val)

{

for (int j = i; j < n - 1; j++)

{

arr[j] = arr[j + 1];

}

}

}

n--;

cout << "Upadated data after deletion is: " << endl;

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

{

cout << arr[i] << endl;

}

}

void main()

{

int arr[5] = { 4, 7, 9, 2, 1 }, n = 5;

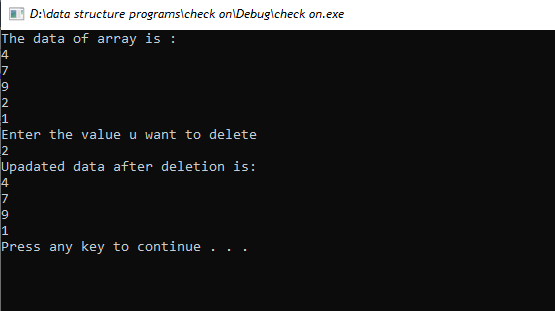
traverse(arr, n);

deletion(arr, n);

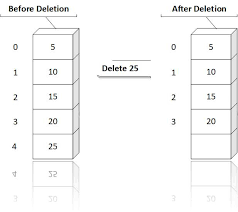
system("pause");

return;

}



* **DELETION\_AT\_END:**



#include "stdafx.h"

#include<iostream>

using namespace std;

void traverse(int arr[], int n)

{

cout << "The data of array is : " << endl;

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

{

cout << arr[i] << endl;;

}

}

void deletion\_at\_end(int arr[], int n)

{

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

{

if (arr[i] == arr[n-1])

{

for (int j = i; j < n - 1; j++)

{

arr[j] = arr[j + 1];

}

}

}

n--;

cout << "Upadated data after deletion is: " << endl;

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

{

cout << arr[i] << endl;

}

}

void main()

{

int arr[5] = { 4, 7, 9, 2, 1 }, n = 5;

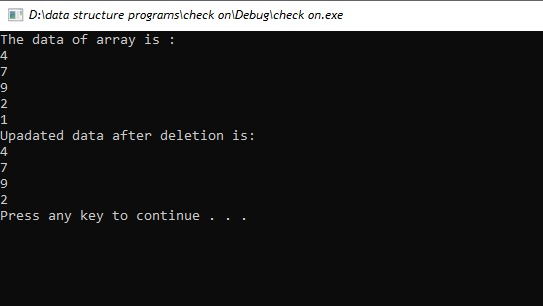
traverse(arr, n);

deletion\_at\_end(arr, n);

system("pause");

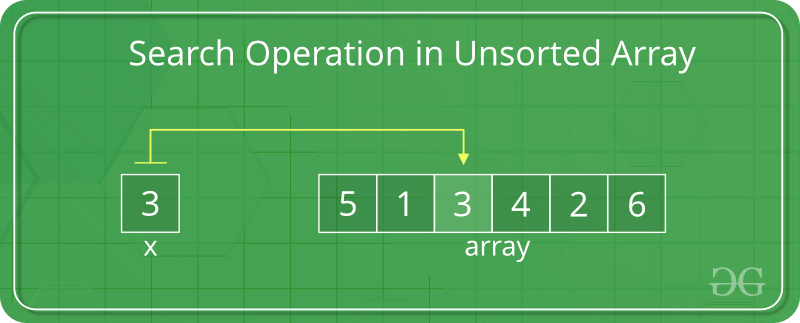
return;

}



1. **SEARCHING:**

In an unsorted array, the search operation can be performed by linear traversal from the first element to the last element.



#include "stdafx.h"

#include<iostream>

using namespace std;

void traverse(int arr[])

{

cout << "The data of array is : " << endl;

for (int i = 0; i < 5; i++)

{

cout << arr[i] << endl;;

}

}

void searching(int arr[])

{

int val, count = 0;

cout << "Enter the value u want to search: " << endl;

cin >> val;

for (int i = 0; i < 5; i++)

{

if (arr[i] == val)

{

cout << "Value founded at " << arr[i] << " location " << endl;

count++;

}

}

if (count == 0)

{

cout << "Value not found" << endl;

}

}

void main()

{

int arr[5] = {4,7,2,8,9};

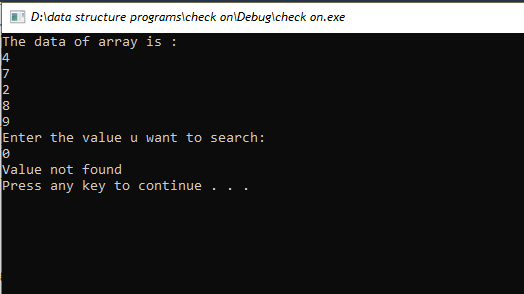
traverse(arr);

searching(arr);

system("pause");

return;

}

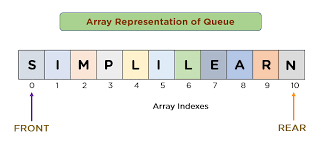


1. **QUEUES USING ARRAYS:**

Queue in array is a data structure that uses an array to store data in a way that follows the first in, first out (FIFO) principle. It allows for the insertion of new elements at the end of the array and the removal of elements from the beginning of the array.

**Representation:**

Following diagram represent the given concept:



1. **TRAVERSING:**

Traversing in queue of array is a process of searching through an array of elements in order to access or modify the data. It involves checking each element in the array sequentially until the desired element is found.

#include "stdafx.h"

#include<iostream>

using namespace std;

int arr[10], n = 5, rear = -1, front = -1;

void display()

{

cout << " LIST OF QUEUE " << endl;

for (int i = front; i <= rear; i++)

{

cout << arr[i] << endl;

}

}

void main()

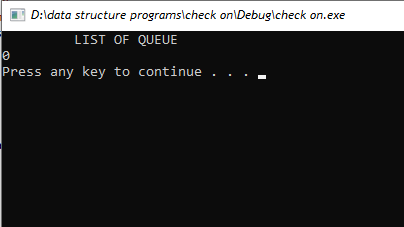
{

display();

system("pause");

return;

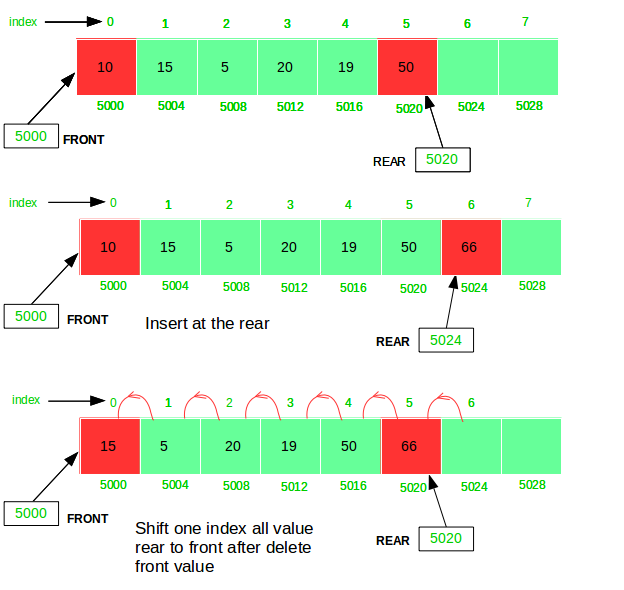
}



1. **INSERTION:**

An insertion in a queue of array is a process that involves adding or inserting an element into an existing queue. The element is added either at the beginning or the end of the queue depending upon the type of queue. The insertion process requires the queue to be shifted in order to make room for the new element.

**Diagramic representation:**



#include "stdafx.h"

#include<iostream>

using namespace std;

int arr[10], n = 5, rear = -1, front = -1;

void display()

{

cout << " LIST OF QUEUE " << endl;

for (int i = front; i <= rear; i++)

{

cout << arr[i] << endl;

}

}

void enqueue()

{

int value;

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

{

if (rear >= n)

{

cout << "QUEUE IS FULL " << endl;

}

else

{

if (front == -1)

{

front++;

}

rear++;

cout << "Enter the value" << endl;

cin >> value;

arr[rear] = value;

}

}

}

void main()

{

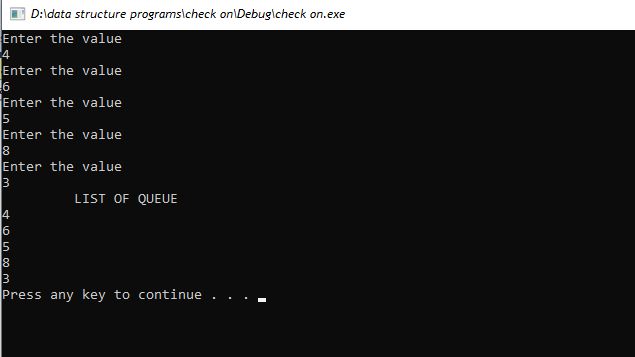
enqueue();

display();

system("pause");

return;

}



1. **DELETION:**

Deletion in a queue of array involves removing an element from the front of the queue. The element is then shifted to the left, so that the next element in the queue becomes the first element. The last element in the queue is then removed from the array.

#include "stdafx.h"

#include<iostream>

using namespace std;

int arr[10], n = 5, rear = -1, front = -1;

void display()

{

cout << " LIST OF QUEUE " << endl;

for (int i = front; i <= rear; i++)

{

cout << arr[i] << endl;

}

}

void dequeue()

{

if (front == -1 && rear == -1)

{

cout << "QUEUE IS EMPTY " << endl;

}

else

{

if (front == rear)

{

front = rear = -1;

}

else{

front++;

}

}

}

void main()

{

display();

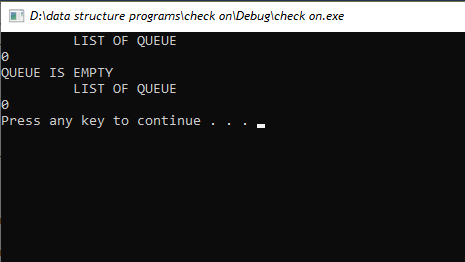
dequeue();

display();

system("pause");

return;

}



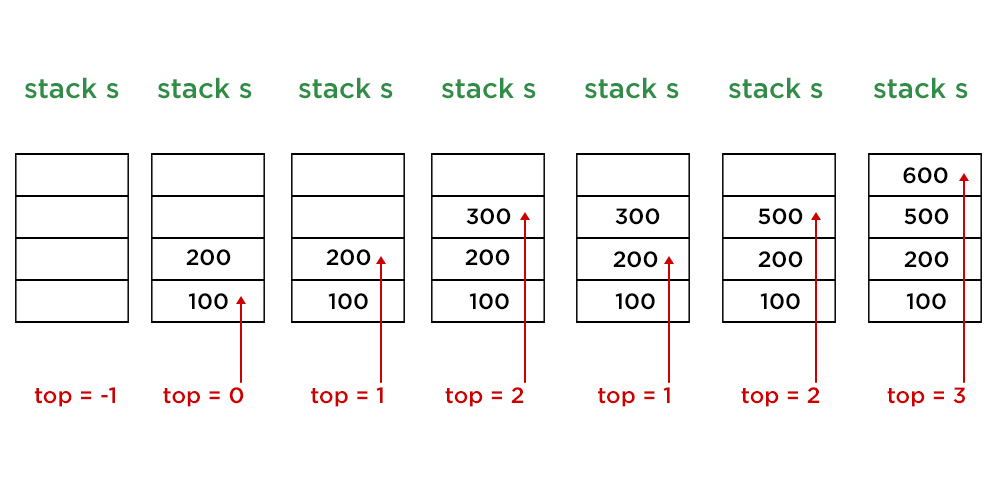
1. **SEARCHING:**

Searching in Queues of Arrays is a process of searching for a specific element in a queue of arrays, which is an ordered group of data stored in memory. The search starts at the beginning of the queue and examines each element in the queue from left to right until the desired element is found.

1. **STACK USING ARRAYS:**

A stack is a data structure that uses a Last-In-First-Out (LIFO) approach to storing and retrieving data. It is implemented using an array, where the last item added to the stack is the first one to be removed. Stacks are often used in computer programming to store and manage data that is used in recursive algorithms.

**Diagramatic representation:**



1. **TRAVERSING:**

Traversal in stack using array is a process of accessing each element in a stack data structure, one at a time, starting from the top of the stack and progressing towards the bottom. The order of element visited is determined by the order in which elements were pushed onto the stack. This traversal is done using an array to store the elements of the stack.

#include "stdafx.h"

#include<iostream>

using namespace std;

class stack{

public:

int n, arr[10], top,value;

stack()

{

top = -1;

n = 5;

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

{

arr[i] = 0;

}

}

bool isempty()

{

if (top == -1)

{

return true;

}

else

{

return false;

}

}

bool isfull()

{

if (top > n)

{

return true;

}

else

{

return false;

}

}

void display()

{

if (isempty())

cout << "STACK UNDERFLOW " << endl;

else

cout << " STACK LIST " << endl;

for (int i = top; i >= 0; i--)

{

cout << arr[i] << endl;

}

}

};

void main()

{

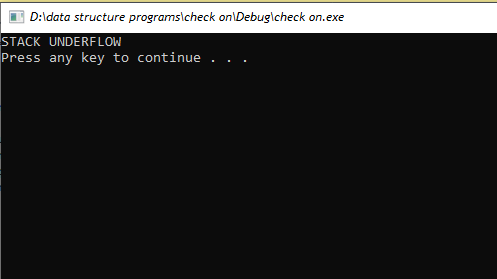
stack s1;

s1.display();

system("pause");

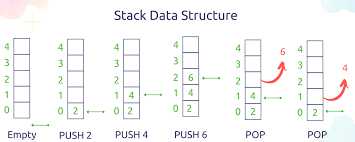
return;

}



1. **INSERTION:**

Insertion in stack using array is the process of pushing (adding) an element onto the top of the stack and allocating the required space in the underlying array to store the new element. This process is performed sequentially and is accompanied by an increase in the size of the stack and the array.



#include "stdafx.h"

#include<iostream>

using namespace std;

class stack{

public:

int n, arr[10], top,value;

stack()

{

top = -1;

n = 5;

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

{

arr[i] = 0;

}

}

bool isempty()

{

if (top == -1)

{

return true;

}

else

{

return false;

}

}

bool isfull()

{

if (top > n)

{

return true;

}

else

{

return false;

}

}

void push()

{

if (isfull())

{

cout << "STACK OVERFLOW " << endl;

}

else

{

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

{

cout << "Enter value " << endl;

cin >> value;

top++;

arr[top] = value;

}

}

}

void display()

{

if (isempty())

cout << "STACK UNDERFLOW " << endl;

else

cout << " STACK LIST " << endl;

for (int i = top; i >= 0; i--)

{

cout << arr[i] << endl;

}

}

};

void main()

{

stack s1;

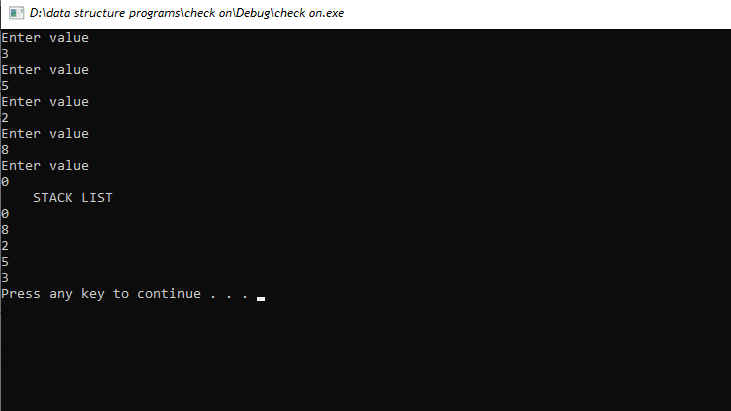
s1.push();

s1.display();

system("pause");

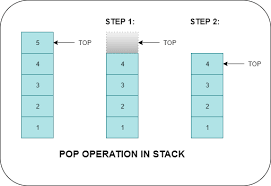
return;

}



1. **DELETION:**

Deletion in a stack implemented using an array is a process of removing the topmost element from the stack and reducing the size of the stack by one. The element that was on the top of the stack is then discarded and the next element in the array becomes the new top element.



#include "stdafx.h"

#include<iostream>

using namespace std;

class stack{

public:

int n, arr[10], top,value;

stack()

{

top = -1;

n = 5;

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

{

arr[i] = 0;

}

}

bool isempty()

{

if (top == -1)

{

return true;

}

else

{

return false;

}

}

bool isfull()

{

if (top > n)

{

return true;

}

else

{

return false;

}

}

void push()

{

if (isfull())

{

cout << "STACK OVERFLOW " << endl;

}

else

{

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

{

cout << "Enter value " << endl;

cin >> value;

top++;

arr[top] = value;

}

}

}

void pop()

{

if (isempty())

{

cout << "Stack overflow" << endl;

}

else

{

top--;

}

}

void display()

{

if (isempty())

cout << "STACK UNDERFLOW " << endl;

else

cout << " STACK LIST " << endl;

for (int i = top; i >= 0; i--)

{

cout << arr[i] << endl;

}

}

};

void main()

{

stack s1;

s1.push();

s1.display();

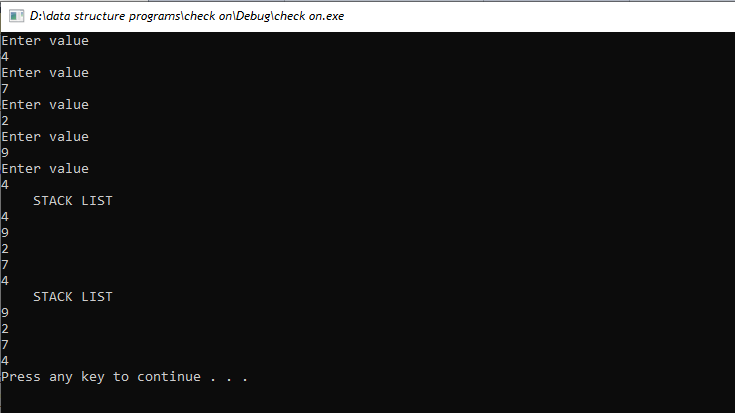
s1.pop();

s1.display();

system("pause");

return;

}

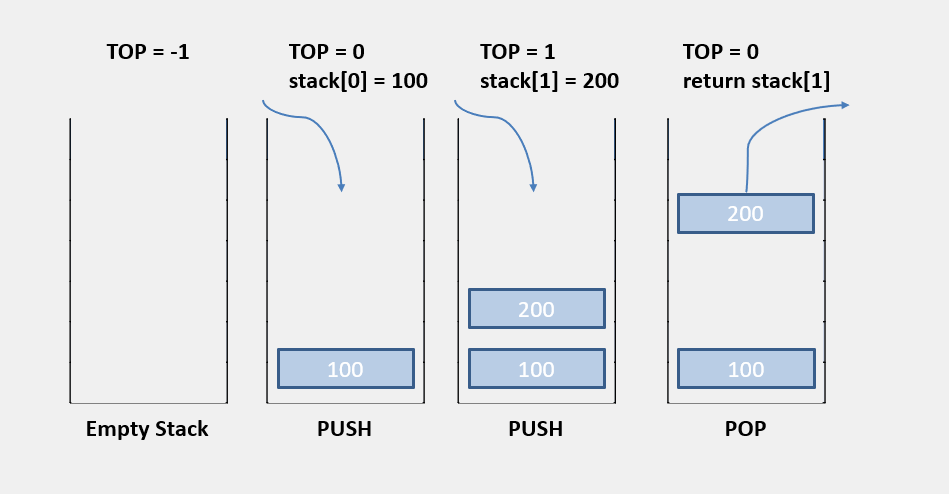


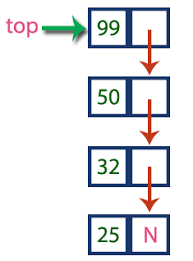
1. **STACK IN LINK-LIST:**

A stack is an abstract data structure that contains a collection of elements. Stack implements the LIFO mechanism i.e. the element that is pushed at the end is popped out first. Some of the principle operations in the stack are −

* Push - This adds a data value to the top of the stack.
* Pop - This removes the data value on top of the stack.
* Peek - This returns the top data value of the stack.

A program that implements a stack using linked list is given as follows





#include "stdafx.h"

#include <iostream>

using namespace std;

struct Node {

int data;

struct Node \*next;

};

struct Node\* top = NULL;

void push(int val) {

struct Node\* newnode = (struct Node\*) malloc(sizeof(struct Node));

newnode->data = val;

newnode->next = top;

top = newnode;

}

void pop() {

if (top == NULL)

cout << "Stack Underflow" << endl;

else {

cout << "The popped element is " << top->data << endl;

top = top->next;

}

}

void display() {

struct Node\* ptr;

if (top == NULL)

cout << "stack is empty";

else {

ptr = top;

cout << "Stack elements are: ";

while (ptr != NULL) {

cout << ptr->data << " ";

ptr = ptr->next;

}

}

cout << endl;

}

int main() {

int ch, val;

cout << "1) Push in stack" << endl;

cout << "2) Pop from stack" << endl;

cout << "3) Display stack" << endl;

cout << "4) Exit" << endl;

do {

cout << "Enter choice: " << endl;

cin >> ch;

switch (ch) {

case 1: {

cout << "Enter value to be pushed:" << endl;

cin >> val;

push(val);

break;

}

case 2: {

pop();

break;

}

case 3: {

display();

break;

}

case 4: {

cout << "Exit" << endl;

break;

}

default: {

cout << "Invalid Choice" << endl;

}

}

} while (ch != 4);

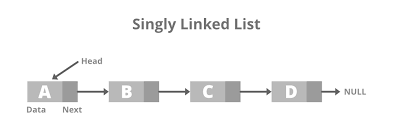
system("pause");

return 0;

}

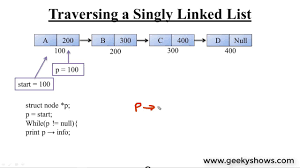
1. **SINGLE LINKLIST :**

A singly linked list is a data structure in which each element points to the next element in the list. It is an ordered collection of nodes, each containing a single data element and a link to the next node in the list. It is a type of linear data structure, and is used in many areas of computer science and other fields.



1. **TRAVERSING:**

Traversal in single linked list is the process of visiting each node of linked list exactly once in order to access data at each node. It can be done by using a loop, which starts from the head node and traverses the list one by one till the last node is reached.



#include "stdafx.h"

#include<iostream>

#include<cstdlib>

using namespace std;

class node{

public:

int data;

node\* next;

};

void traverse(node \*\*head)

{

node \*temp= \*head;

while (temp != NULL)

{

cout << temp->data << endl;

temp = temp->next;

}

}

void main()

{

node \*head;

node \*one ;

node \*two = NULL;

node \*three = NULL;

node \*four = NULL;

one = new node();

two = new node();

three = new node();

four = new node();

one->data = 1;

one->next = two;

two->data = 2;

two->next = three;

three->data = 3;

three->next = four;

four->data = 4;

four->next = NULL;

head = one;

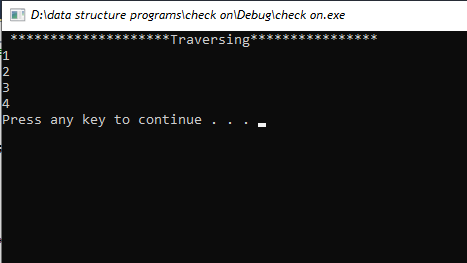
cout<< " \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Traversing\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* " << endl;

traverse(&head);

system("pause");

return;

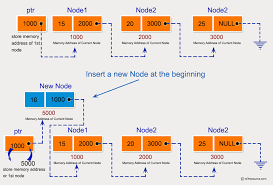
}



1. **INSERTION:**

* **INSERTION\_AT\_START:**

Insertion at the start of a singly linked list involves creating a new node, setting the new node's next pointer to the head of the existing list, and setting the head of the list to the new node. This operation is relatively efficient, as it requires only one link traversal.



#include "stdafx.h"

#include<iostream>

#include<cstdlib>

using namespace std;

class node{

public:

int data;

node\* next;

};

void traverse(node \*\*head)

{

node \*temp= \*head;

while (temp != NULL)

{

cout << temp->data << endl;

temp = temp->next;

}

}

void insert\_at\_start(node \*\*head)

{

node \*newnode = NULL;

newnode = new node();

newnode->next = \*head;

\*head = newnode;

newnode->data = 0;

}

void main()

{

node \*head;

node \*one ;

node \*two = NULL;

node \*three = NULL;

node \*four = NULL;

one = new node();

two = new node();

three = new node();

four = new node();

one->data = 1;

one->next = two;

two->data = 2;

two->next = three;

three->data = 3;

three->next = four;

four->data = 4;

four->next = NULL;

head = one;

cout << " \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Insertion at start\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* " << endl;

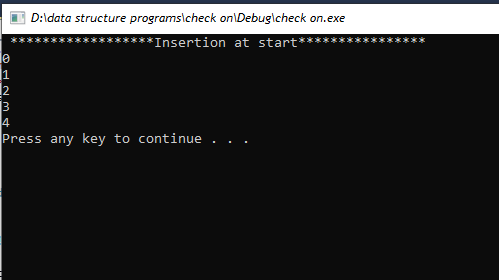
insert\_at\_start(&head);

traverse(&head);

system("pause");

return;

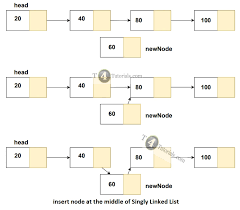
}



:

* **INSERTION\_AT\_MIDDLE:**

Insertion at mid in single linklist is a process of inserting a new node at the middle of the existing linked list. It needs to traverse the list to find the middle element and then insert the new node after the middle element.



#include "stdafx.h"

#include<iostream>

#include<cstdlib>

using namespace std;

class node{

public:

int data;

node\* next;

};

void traverse(node \*\*head)

{

node \*temp= \*head;

while (temp != NULL)

{

cout << temp->data << endl;

temp = temp->next;

}

}

void insert\_at\_middle(node \*\*head, int key)

{

node \*newnode = NULL;

newnode = new node();

node \*temp = \*head;

node \*temp1;

for (int i = 1; i < key;i++)

{

temp = temp->next;

}

temp1 = temp->next;

temp->next = newnode;

newnode->next = temp1;

newnode->data = 0;

return;

}

void main()

{

node \*head;

node \*one ;

node \*two = NULL;

node \*three = NULL;

node \*four = NULL;

one = new node();

two = new node();

three = new node();

four = new node();

one->data = 1;

one->next = two;

two->data = 2;

two->next = three;

three->data = 3;

three->next = four;

four->data = 4;

four->next = NULL;

head = one;

int key;

cout << " Enter the position where u want to insert a node " << endl;

cin >> key;

cout << " \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Inertion at middle\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* " << endl;

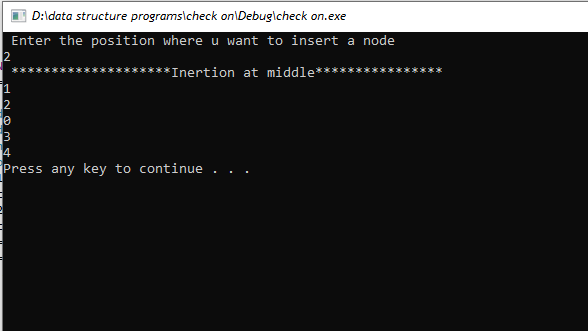
insert\_at\_middle(&head, key);

traverse(&head);

system("pause");

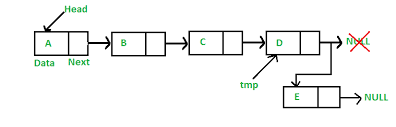
return;

}



* **INSERTION\_AT\_END:**

Insertion at the end of a single linked list involves creating a new node, assigning it the data to be inserted, setting its link to NULL, then traversing the list to find the last node whose link is not NULL and setting its link to the new node.



#include "stdafx.h"

#include<iostream>

#include<cstdlib>

using namespace std;

class node{

public:

int data;

node\* next;

};

void traverse(node \*\*head)

{

node \*temp= \*head;

while (temp != NULL)

{

cout << temp->data << endl;

temp = temp->next;

}

}

void insert\_at\_end(node \*\*head)

{

node \*newnode = NULL;

newnode = new node();

node \*temp = \*head;

while (temp->next != NULL)

{

temp = temp->next;

}

temp->next = newnode;

newnode->next = NULL;

newnode->data = 25;

}

void main()

{

node \*head;

node \*one ;

node \*two = NULL;

node \*three = NULL;

node \*four = NULL;

one = new node();

two = new node();

three = new node();

four = new node();

one->data = 1;

one->next = two;

two->data = 2;

two->next = three;

three->data = 3;

three->next = four;

four->data = 4;

four->next = NULL;

head = one;

cout << " \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Insertion at end\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* " << endl;

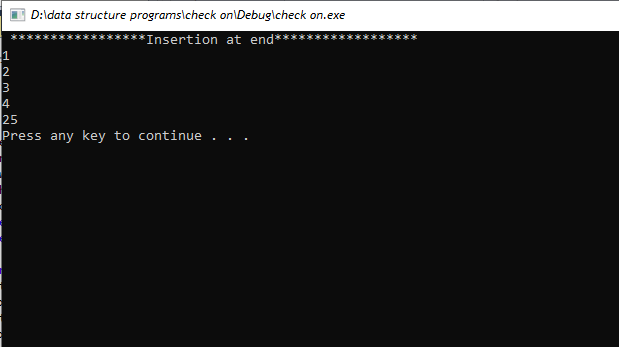
insert\_at\_end(&head);

traverse(&head);

system("pause");

return;

}



1. **UPDATION:**

Updating an element in a singly linked list involves replacing the value of an existing node with a new value. The update operation can be implemented by traversing the list, locating the node to be updated, and then replacing the existing value with the new one.

* **UPDATION\_AT\_START:**

#include "stdafx.h"

#include<iostream>

#include<cstdlib>

using namespace std;

class node{

public:

int data;

node\* next;

};

void traverse(node \*\*head)

{

node \*temp= \*head;

while (temp != NULL)

{

cout << temp->data << endl;

temp = temp->next;

}

}

void updation\_at\_start(node \*\*head,int value)

{

node \*temp;

temp = \*head;

temp->data = value;

\*head = temp;

}

void main()

{

node \*head;

node \*one ;

node \*two = NULL;

node \*three = NULL;

node \*four = NULL;

one = new node();

two = new node();

three = new node();

four = new node();

one->data = 1;

one->next = two;

two->data = 2;

two->next = three;

three->data = 3;

three->next = four;

four->data = 4;

four->next = NULL;

head = one;

cout << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*UPDATION AT START\*\*\*\*\*\*\*\*\*\*\*\*\*" << endl;

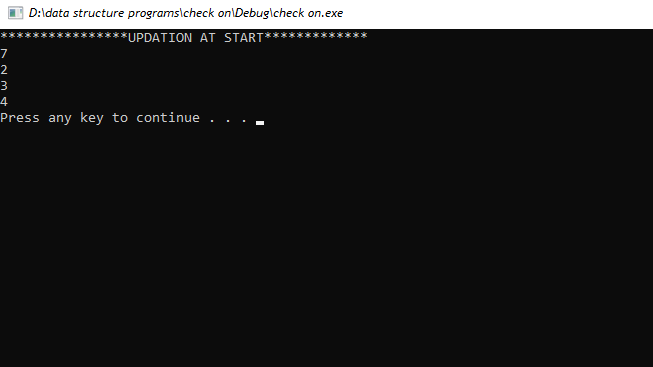
updation\_at\_start(&head, 7);

traverse(&head);

system("pause");

return;

}



* **UPDATION\_AT\_MIDDLE:**

#include "stdafx.h"

#include<iostream>

#include<cstdlib>

using namespace std;

class node{

public:

int data;

node\* next;

};

void traverse(node \*\*head)

{

node \*temp= \*head;

while (temp != NULL)

{

cout << temp->data << endl;

temp = temp->next;

}

}

void updation\_at\_middle(node \*\*head, int value)

{

int pos;

cout << " Enter the position where u want updation " << endl;

cin >> pos;

node \*temp;

temp = \*head;

for (int i = 1; i < pos; i++)

{

temp = temp->next;

}

temp->data = value;

}

void main()

{

node \*head;

node \*one ;

node \*two = NULL;

node \*three = NULL;

node \*four = NULL;

one = new node();

two = new node();

three = new node();

four = new node();

one->data = 1;

one->next = two;

two->data = 2;

two->next = three;

three->data = 3;

three->next = four;

four->data = 4;

four->next = NULL;

head = one;

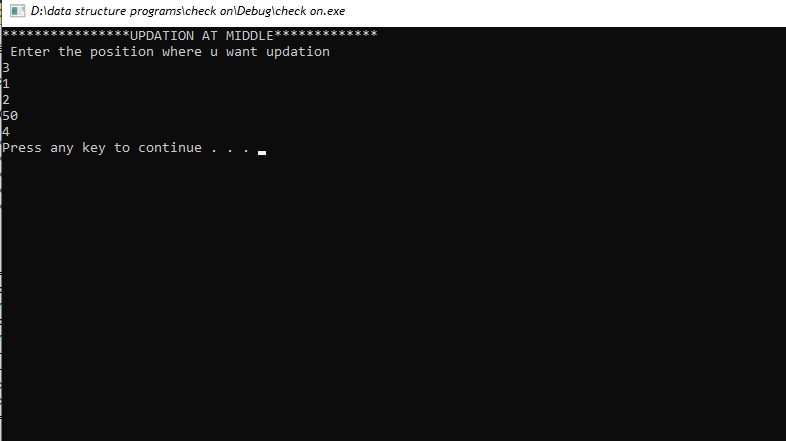
cout << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*UPDATION AT MIDDLE\*\*\*\*\*\*\*\*\*\*\*\*\*" << endl;

updation\_at\_middle(&head, 50);

traverse(&head); system("pause");

return;

}



* **UPDATION\_AT\_END:**

#include "stdafx.h"

#include<iostream>

#include<cstdlib>

using namespace std;

class node{

public:

int data;

node\* next;

};

void traverse(node \*\*head)

{

node \*temp= \*head;

while (temp != NULL)

{

cout << temp->data << endl;

temp = temp->next;

}

}

void updation\_at\_end(node \*\*head, int value)

{

node \*temp;

temp = \*head;

while (temp->next != NULL)

{

temp = temp->next;

}

temp->data = value;

}

void main()

{

node \*head;

node \*one ;

node \*two = NULL;

node \*three = NULL;

node \*four = NULL;

one = new node();

two = new node();

three = new node();

four = new node();

one->data = 1;

one->next = two;

two->data = 2;

two->next = three;

three->data = 3;

three->next = four;

four->data = 4;

four->next = NULL;

head = one;

cout << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*UPDATION AT END\*\*\*\*\*\*\*\*\*\*\*\*\*" << endl;

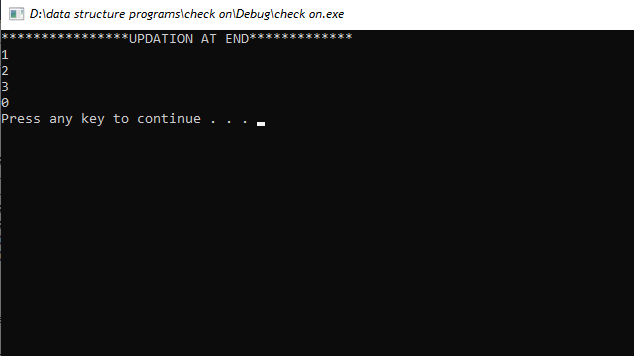
updation\_at\_end(&head, 0);

traverse(&head);

system("pause");

return;

}

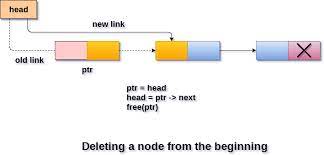


1. **DELETION:**

Deletion in a single linked list is the process of removing a node from a linked list. It involves adjusting the links that point to the node to be deleted so that the previous node now points to the node after the deleted node. Deletion can occur at the start, end, or in the middle of the list.

* **DELETION\_AT\_START:**

For deletion element at start,simply create a new link between head and 2nd node.delete the first node



#include "stdafx.h"

#include<iostream>

#include<cstdlib>

using namespace std;

class node{

public:

int data;

node\* next;

};

void traverse(node \*\*head)

{

node \*temp= \*head;

while (temp != NULL)

{

cout << temp->data << endl;

temp = temp->next;

}

}

void deletion\_at\_start(node \*\*head)

{

node \*temp = \*head;

\*head =temp->next;

delete (temp);

}

void main()

{

node \*head;

node \*one ;

node \*two = NULL;

node \*three = NULL;

node \*four = NULL;

one = new node();

two = new node();

three = new node();

four = new node();

one->data = 1;

one->next = two;

two->data = 2;

two->next = three;

three->data = 3;

three->next = four;

four->data = 4;

four->next = NULL;

head = one;

cout << "Linklist before deletion :" << endl;

traverse(&head);

cout << " \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Deletion at start\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* " << endl;

deletion\_at\_start(&head);

traverse(&head);

system("pause");

return;

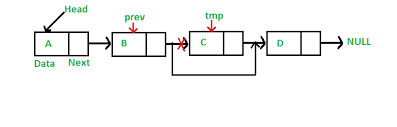
}



* **DELETION\_AT\_MIDDLE:**

Deletion at mid in a single linked list is a process of removing a node from the middle of the linked list. It involves changing the links of the nodes adjacent to the node to be deleted in order to remove the node from the list. This process requires the node to be deleted to be located first.

**DIagramatic representation:**



#include "stdafx.h"

#include<iostream>

#include<cstdlib>

using namespace std;

class node{

public:

int data;

node\* next;

};

void traverse(node \*\*head)

{

node \*temp= \*head;

while (temp != NULL)

{

cout << temp->data << endl;

temp = temp->next;

}

}

void deletion\_at\_middle(node \*\*head,int key)

{

node\*temp = \*head;

for (int i = 1; i < key-1; i++)

{

temp = temp->next;

}

node \*d = temp->next;

temp->next = d->next;

delete(d);

return;

}

void main()

{

node \*head;

node \*one ;

node \*two = NULL;

node \*three = NULL;

node \*four = NULL;

one = new node();

two = new node();

three = new node();

four = new node();

one->data = 1;

one->next = two;

two->data = 2;

two->next = three;

three->data = 3;

three->next = four;

four->data = 4;

four->next = NULL;

head = one;

cout << "Linklist before deletion :" << endl;

traverse(&head);

int key;

cout << "Enter the location to node u want to delete" << endl;

cin >> key;

cout << " \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Deletion at middle\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* " << endl;

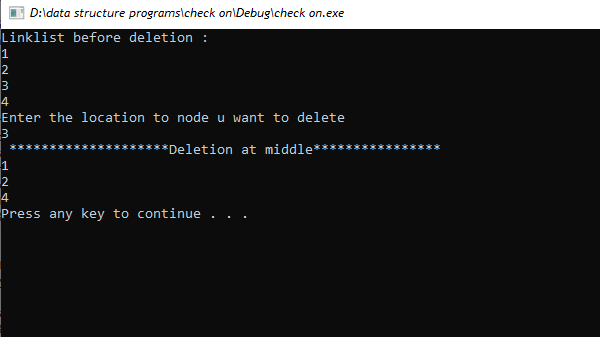
deletion\_at\_middle(&head,key);

traverse(&head);

system("pause");

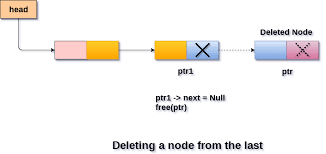
return;

}



* **DELETION\_AT\_END:**

Deletion at the end of a single linked list means removing the last node in the list. This involves traversing the list to find the second-to-last node and then updating its next pointer to null. This operation requires O(n) time complexity.



#include "stdafx.h"

#include<iostream>

#include<cstdlib>

using namespace std;

class node{

public:

int data;

node\* next;

};

void traverse(node \*\*head)

{

node \*temp= \*head;

while (temp != NULL)

{

cout << temp->data << endl;

temp = temp->next;

}

}

void deletion\_at\_end(node \*\*head)

{

node\*temp = \*head;

node\*per=NULL;

while (temp ->next!= NULL)

{

per = temp;

temp = temp->next;

}

per->next = NULL;

delete (temp);

return;

}

void main()

{

node \*head;

node \*one ;

node \*two = NULL;

node \*three = NULL;

node \*four = NULL;

one = new node();

two = new node();

three = new node();

four = new node();

one->data = 1;

one->next = two;

two->data = 2;

two->next = three;

three->data = 3;

three->next = four;

four->data = 4;

four->next = NULL;

head = one;

cout << "Linklist before deletion :" << endl;

traverse(&head);

int key;

cout << " \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Deletion at end\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* " << endl;

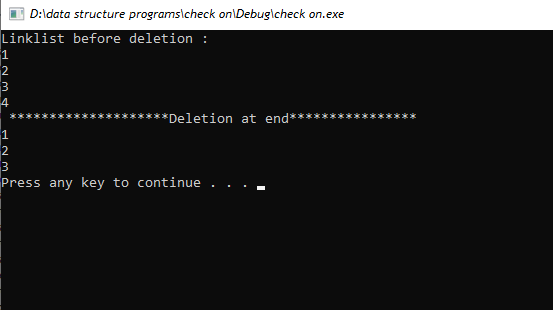
deletion\_at\_end(&head);

traverse(&head);

system("pause");

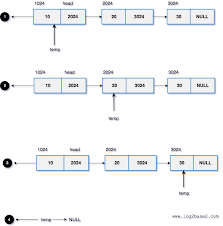
return;

}



1. **SEARCHING:**

Searching in a single linked list is the process of scanning through the list to find a particular item or data. It involves comparing each node's data with the given data to find the desired item. Searching in a single linked list is done sequentially and takes O(n) time in the worst case, where n is the length of the linked list.



#include "stdafx.h"

#include<iostream>

#include<cstdlib>

using namespace std;

class node{

public:

int data;

node\* next;

};

void traverse(node \*\*head)

{

node \*temp= \*head;

while (temp != NULL)

{

cout << temp->data << endl;

temp = temp->next;

}

}

void searching(node \*\*head)

{

int value,count=0;

cout << " Enter the value u want to search " << endl;

cin >> value;

node \*temp;

temp = \*head;

while (temp!= NULL)

{

if (temp->data==value)

{

cout << " value founded " << temp->data << endl;

count++;

}

temp = temp->next;

}

if (count == 0)

{

cout << "Value not founded " << endl;

}

}

void main()

{

node \*head;

node \*one ;

node \*two = NULL;

node \*three = NULL;

node \*four = NULL;

one = new node();

two = new node();

three = new node();

four = new node();

one->data = 1;

one->next = two;

two->data = 2;

two->next = three;

three->data = 3;

three->next = four;

four->data = 4;

four->next = NULL;

head = one;

int key;

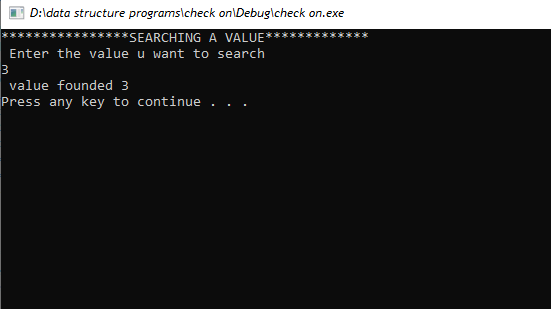
cout << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*SEARCHING A VALUE\*\*\*\*\*\*\*\*\*\*\*\*\*" << endl;

searching(&head);

system("pause");

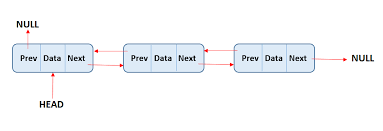
return;

}



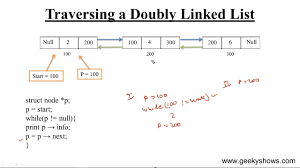
1. **SINGLE DOUBLE LINKLIST:**

A singly linked list is a linear data structure in which each node contains a pointer to the next node in the list. A doubly linked list is a more complex type of linked list in which each node contains a pointer to the next node as well as a pointer to the previous node. This allows for a more efficient traversal of the data structure in both directions.



1. **TRAVERSING:**

Traversal of a double linked list is the process of visiting each node in the list, starting from the head node and following the links to the next node until it reaches the last node. Traversal allows for the manipulation of data in the list and can be done in either a forward or backward direction.



#include "stdafx.h"

#include<iostream>

using namespace std;

class node {

public:

int data;

node\* next;

node\* prev;

};

void traverse(node\*\* head)

{

{

node\* temp = \*head;

while (temp != NULL) {

cout << temp->data << endl;

temp = temp->next;

}

}

}

int main()

{

{

node\* head;

node\* one = NULL;

node\* two = NULL;

node\* three = NULL;

node\* four = NULL;

one = new node();

two = new node();

three = new node();

four = new node();

one->data = 1;

one->next = two;

one->prev = NULL;

two->data = 2;

two->next = three;

two->prev = one;

three->data = 3;

three->next = four;

three->prev = two;

four->data = 4;

four->next = NULL;

four->prev = three;

head = one;

cout << " LIST " << endl;

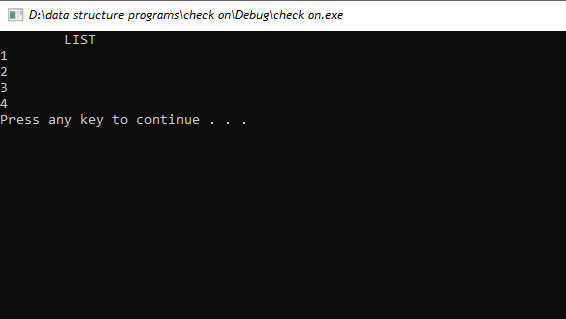
traverse(&head);

system("pause");

return 0;

}

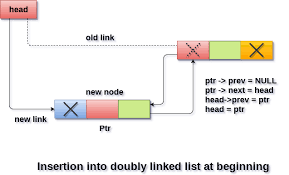
}



1. **INSERTION:**

* **INSERTION\_AT\_START:**

Insertion at start of a doubly linked list is a process of inserting a new node at the beginning of the list. This operation requires the node to be inserted to point to the head node and the head node to point to the new node. This operation is done in constant time, O(1).



#include "stdafx.h"

#include<iostream>

using namespace std;

class node {

public:

int data;

node\* next;

node\* prev;

};

void traverse(node\*\* head)

{

{

node\* temp = \*head;

while (temp != NULL) {

cout << temp->data << endl;

temp = temp->next;

}

}

}

void insert\_at\_start(node\*\* head)

{

{

node\* temp = \*head;

node\* newnode = NULL;

newnode = new node();

newnode->data = 7;

newnode->next = temp;

newnode->prev = NULL;

temp->prev = newnode;

\*head = newnode;

}

}

int main()

{

{

node\* head;

node\* one = NULL;

node\* two = NULL;

node\* three = NULL;

node\* four = NULL;

one = new node();

two = new node();

three = new node();

four = new node();

one->data = 1;

one->next = two;

one->prev = NULL;

two->data = 2;

two->next = three;

two->prev = one;

three->data = 3;

three->next = four;

three->prev = two;

four->data = 4;

four->next = NULL;

four->prev = three;

head = one;

cout << " INSERTION AT START " << endl;

insert\_at\_start(&head);

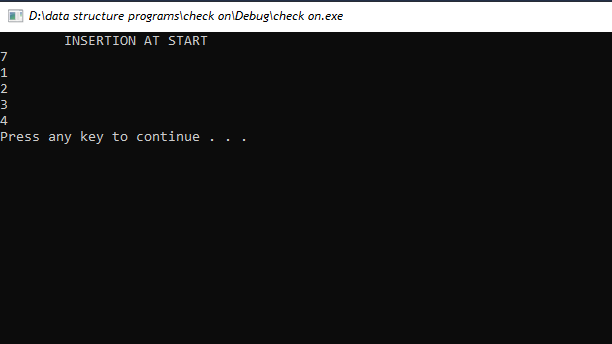
traverse(&head);

system("pause");

return 0;

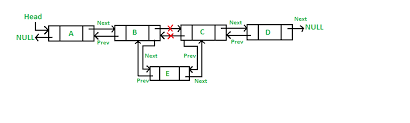
}

}



* **INSERTION\_AT\_MIDDLE**

Insertion at the middle of a doubly linked list is a process of inserting a new node between two existing nodes in the list. This is done by making the node's next pointer point to the node after the new node, and the node's previous pointer point to the node before the new node. The new node's next and previous pointers are then set to point to the nodes before and after it.



#include "stdafx.h"

#include<iostream>

using namespace std;

class node {

public:

int data;

node\* next;

node\* prev;

};

void traverse(node\*\* head)

{

{

node\* temp = \*head;

while (temp != NULL) {

cout << temp->data << endl;

temp = temp->next;

}

}

}

void insert\_at\_middle(node\*\* head, int pos)

{

node\* temp = \*head;

node\* midnode = NULL;

midnode = new node();

for (int i = 1; i<pos - 1; i++)

{

temp = temp->next;

}

midnode->data = 7;

midnode->next = temp->next;;

temp->next = midnode;

midnode->prev = temp;

}

int main()

{

node\* head;

node\* one = NULL;

node\* two = NULL;

node\* three = NULL;

node\* four = NULL;

one = new node();

two = new node();

three = new node();

four = new node();

one->data = 1;

one->next = two;

one->prev = NULL;

two->data = 2;

two->next = three;

two->prev = one;

three->data = 3;

three->next = four;

three->prev = two;

four->data = 4;

four->next = NULL;

four->prev = three;

head = one;

cout << "Linklist before Insertion :" << endl;

traverse(&head);

int pos;

cout << "Enter position u want to insert node " << endl;

cin >> pos;

cout << " INSERTION AT MIDDLE " << endl;

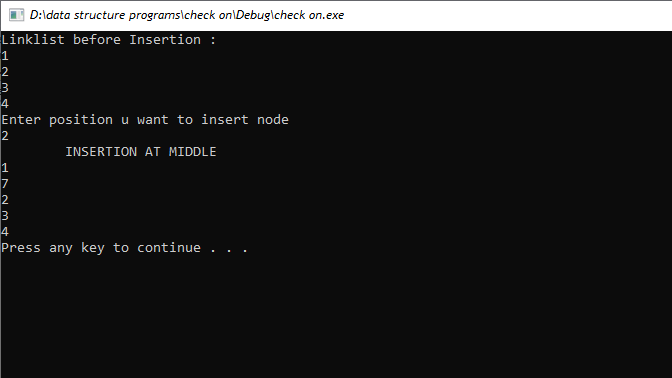
insert\_at\_middle(&head, pos);

traverse(&head);

system("pause");

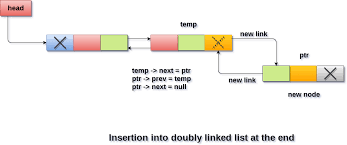
return 0;

}



* **INSERTION\_AT\_END:**

Insertion at the end of a double linked list is the process of adding a new node at the end of the list. The new node will have its previous pointer set to the last node of the list and its next pointer set to null. This operation is generally used when new elements need to be added to the end of the list.



#include "stdafx.h"

#include<iostream>

using namespace std;

class node {

public:

int data;

node\* next;

node\* prev;

};

void traverse(node\*\* head)

{

{

node\* temp = \*head;

while (temp != NULL) {

cout << temp->data << endl;

temp = temp->next;

}

}

}

void insert\_at\_end(node\*\* head)

{

node\* temp = \*head;

node\* endnode = NULL;

endnode = new node();

while (temp->next != NULL)

{

temp = temp->next;

}

endnode->data = 7;

endnode->next = NULL;

endnode->prev = temp;

temp->next = endnode;

}

int main()

{

node\* head;

node\* one = NULL;

node\* two = NULL;

node\* three = NULL;

node\* four = NULL;

one = new node();

two = new node();

three = new node();

four = new node();

one->data = 1;

one->next = two;

one->prev = NULL;

two->data = 2;

two->next = three;

two->prev = one;

three->data = 3;

three->next = four;

three->prev = two;

four->data = 4;

four->next = NULL;

four->prev = three;

head = one;

cout << "Linklist before Insertion :" << endl;

traverse(&head);

cout << " INSERTION AT END " << endl;

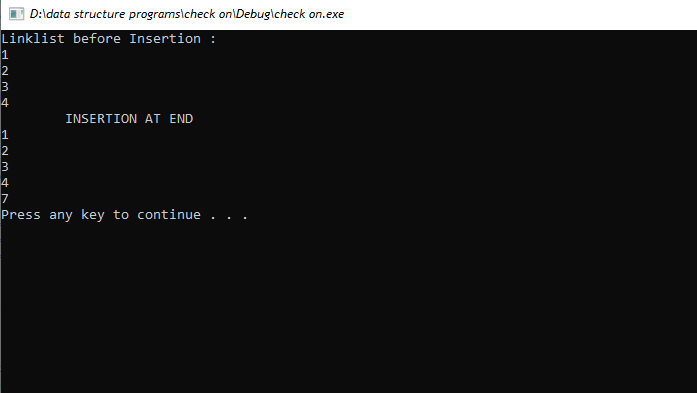
insert\_at\_end(&head);

traverse(&head);

system("pause");

return 0;

}



1. **UPDATION:**

Updation in double linked list is the process of modifying the existing data in the double linked list. It involves locating the node which needs to be modified and then updating the data with the new data. This can be done by traversing the list till the desired node is found. Then the data is updated and the list is re-arranged accordingly.

* **UPDATION\_At\_START**

#include "stdafx.h"

#include<iostream>

using namespace std;

class node {

public:

int data;

node\* next;

node\* prev;

};

void traverse(node\*\* head)

{

{

node\* temp = \*head;

while (temp != NULL) {

cout << temp->data << endl;

temp = temp->next;

}

}

}

void updation\_at\_start(node \*\*head,int x)

{

node \*temp = \*head;

temp->data = x;

}

int main()

{

node\* head;

node\* one = NULL;

node\* two = NULL;

node\* three = NULL;

node\* four = NULL;

one = new node();

two = new node();

three = new node();

four = new node();

one->data = 1;

one->next = two;

one->prev = NULL;

two->data = 2;

two->next = three;

two->prev = one;

three->data = 3;

three->next = four;

three->prev = two;

four->data = 4;

four->next = NULL;

four->prev = three;

head = one;

cout << "Linklist before Updation :" << endl;

traverse(&head);

cout << " UPDATION AT START " << endl;

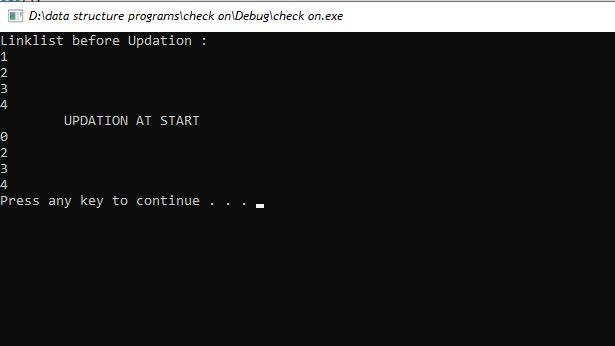
updation\_at\_start(&head,0);

traverse(&head);

system("pause");

return 0;

}



* **UPDATION\_At\_MIDDLE:**

#include "stdafx.h"

#include<iostream>

using namespace std;

class node {

public:

int data;

node\* next;

node\* prev;

};

void traverse(node\*\* head)

{

{

node\* temp = \*head;

while (temp != NULL) {

cout << temp->data << endl;

temp = temp->next;

}

}

}

void updation\_at\_middle(node \*\*head, int x)

{

int pos;

cout << "Enter the postion u want to update " << endl;

cin >> pos;

node \*temp = \*head;

for (int i = 1; i < pos;i++)

{

temp = temp->next;

}

temp->data = x;

}

int main()

{

node\* head;

node\* one = NULL;

node\* two = NULL;

node\* three = NULL;

node\* four = NULL;

one = new node();

two = new node();

three = new node();

four = new node();

one->data = 1;

one->next = two;

one->prev = NULL;

two->data = 2;

two->next = three;

two->prev = one;

three->data = 3;

three->next = four;

three->prev = two;

four->data = 4;

four->next = NULL;

four->prev = three;

head = one;

cout << "Linklist before Updation :" << endl;

traverse(&head);

cout << " UPDATION AT MIDDLE " << endl;

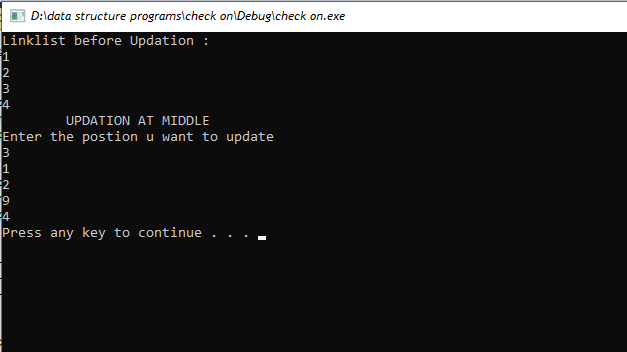
updation\_at\_middle(&head, 9);

traverse(&head);

system("pause");

return 0;

}



* **UPDATION\_At\_END:**

#include "stdafx.h"

#include<iostream>

using namespace std;

class node {

public:

int data;

node\* next;

node\* prev;

};

void traverse(node\*\* head)

{

{

node\* temp = \*head;

while (temp != NULL) {

cout << temp->data << endl;

temp = temp->next;

}

}

}

void updation\_at\_end(node \*\*head, int x)

{

node \*temp = \*head;

while (temp->next != NULL)

{

temp = temp->next;

}

temp->data = x;

}

int main()

{

node\* head;

node\* one = NULL;

node\* two = NULL;

node\* three = NULL;

node\* four = NULL;

one = new node();

two = new node();

three = new node();

four = new node();

one->data = 1;

one->next = two;

one->prev = NULL;

two->data = 2;

two->next = three;

two->prev = one;

three->data = 3;

three->next = four;

three->prev = two;

four->data = 4;

four->next = NULL;

four->prev = three;

head = one;

cout << "Linklist before Updation :" << endl;

traverse(&head);

cout << " UPDATION AT END " << endl;

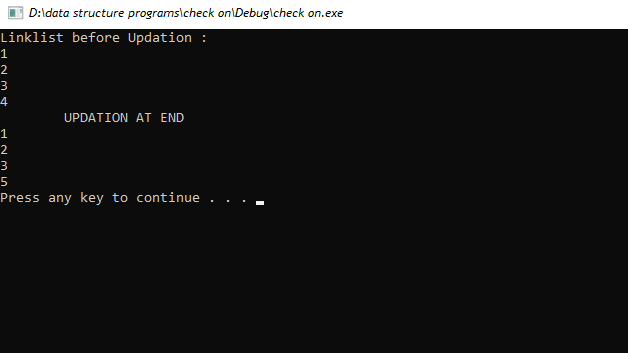
updation\_at\_end(&head, 5);

traverse(&head);

system("pause");

return 0;

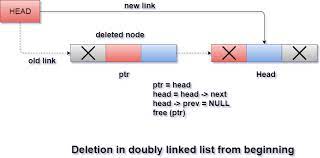
}



1. **DELETION:**

Deletion in double linked list is the process of removing a node from the list. It involves setting the previous node's next pointer to the deleted node's next pointer and setting the next node's previous pointer to the deleted node's previous pointer. This ensures that the deleted node is removed from the list and the rest of the list remains connected.

* **DELETION\_AT\_START:**



#include "stdafx.h"

#include<iostream>

using namespace std;

class node {

public:

int data;

node\* next;

node\* prev;

};

void traverse(node\*\* head)

{

{

node\* temp = \*head;

while (temp != NULL) {

cout << temp->data << endl;

temp = temp->next;

}

}

}

void deletion\_at\_start(node \*\*head)

{

node \*temp = \*head;

temp->next->prev = NULL;

\*head=temp->next;

delete(temp);

}

int main()

{

node\* head;

node\* one = NULL;

node\* two = NULL;

node\* three = NULL;

node\* four = NULL;

one = new node();

two = new node();

three = new node();

four = new node();

one->data = 1;

one->next = two;

one->prev = NULL;

two->data = 2;

two->next = three;

two->prev = one;

three->data = 3;

three->next = four;

three->prev = two;

four->data = 4;

four->next = NULL;

four->prev = three;

head = one;

cout << "Linklist before Deletion :" << endl;

traverse(&head);

cout << " DELETION AT START " << endl;

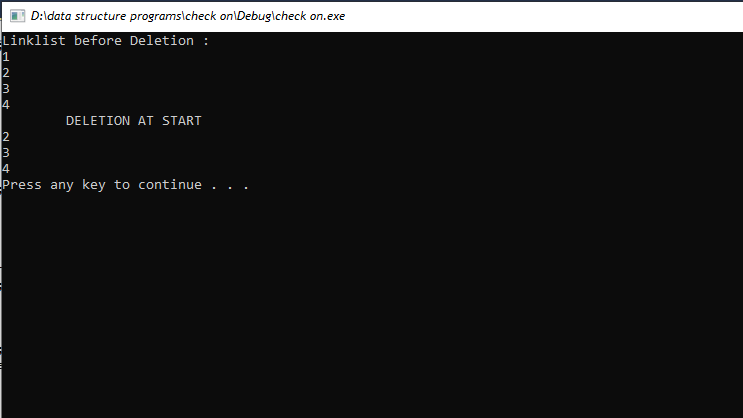
deletion\_at\_start(&head);

traverse(&head);

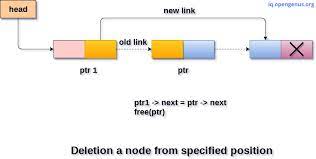
system("pause");

return 0;

}



* **DELETION\_AT\_MIDDLE:**



#include "stdafx.h"

#include<iostream>

using namespace std;

class node {

public:

int data;

node\* next;

node\* prev;

};

void traverse(node\*\* head)

{

{

node\* temp = \*head;

while (temp != NULL) {

cout << temp->data << endl;

temp = temp->next;

}

}

}

void deletion\_at\_middle(node \*\*head)

{

int pos;

cout << "Enter the position u want to delete" << endl;

cin >> pos;

node \*temp = \*head;

for (int i = 1; i < pos; i++)

{

temp = temp->next;

}

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

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

delete(temp);

}int main()

{

node\* head;

node\* one = NULL;

node\* two = NULL;

node\* three = NULL;

node\* four = NULL;

one = new node();

two = new node();

three = new node();

four = new node();

one->data = 1;

one->next = two;

one->prev = NULL;

two->data = 2;

two->next = three;

two->prev = one;

three->data = 3;

three->next = four;

three->prev = two;

four->data = 4;

four->next = NULL;

four->prev = three;

head = one;

cout << "Linklist before Deletion :" << endl;

traverse(&head);

cout << " DELETION AT MIDDLE " << endl;

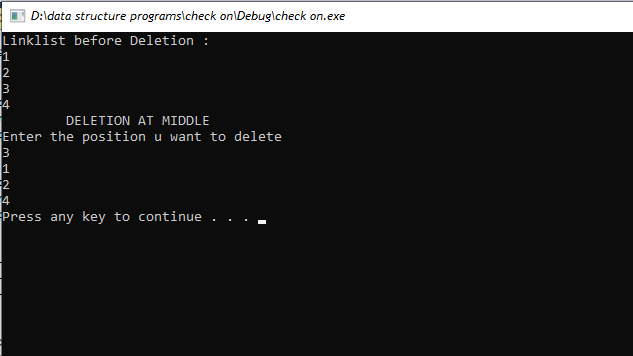
deletion\_at\_middle(&head);

traverse(&head);

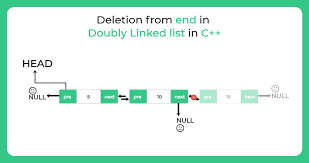
system("pause");

return 0;

}



* **DELETION\_AT\_END:**



#include "stdafx.h"

#include<iostream>

using namespace std;

class node {

public:

int data;

node\* next;

node\* prev;

};

void traverse(node\*\* head)

{

{

node\* temp = \*head;

while (temp != NULL) {

cout << temp->data << endl;

temp = temp->next;

}

}

}

void deletion\_at\_end(node \*\*head)

{

node \*temp = \*head;

while (temp->next != NULL)

{

temp = temp->next;

}

temp->prev->next = NULL;

delete(temp);

}

int main()

{

node\* head;

node\* one = NULL;

node\* two = NULL;

node\* three = NULL;

node\* four = NULL;

one = new node();

two = new node();

three = new node();

four = new node();

one->data = 1;

one->next = two;

one->prev = NULL;

two->data = 2;

two->next = three;

two->prev = one;

three->data = 3;

three->next = four;

three->prev = two;

four->data = 4;

four->next = NULL;

four->prev = three;

head = one;

cout << "Linklist before Deletion :" << endl;

traverse(&head);

cout << " DELETION AT END " << endl;

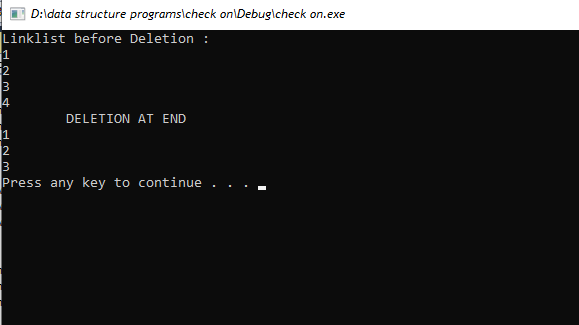
deletion\_at\_end(&head);

traverse(&head);

system("pause");

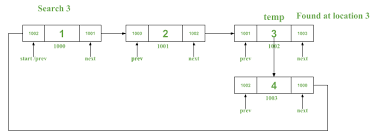
return 0;

}



1. **SEARCHING:**

Searching in a double linked list is a process where elements of the list can be quickly located by traversing through the list. It is done by comparing each element of the list to the desired element. If a match is found, then the element is returned. Otherwise, the search continues until a match is found or the end of the list is reached.



#include "stdafx.h"

#include<iostream>

using namespace std;

class node {

public:

int data;

node\* next;

node\* prev;

};

void traverse(node\*\* head)

{

{

node\* temp = \*head;

while (temp != NULL) {

cout << temp->data << endl;

temp = temp->next;

}

}

}

void searching(node \*\*head)

{

int a, count = 0;

cout << "Enter number you wanna search:" << endl;

cin >> a;

node\*temp = \*head;

do{

if (temp->data == a)

{

cout << "The value found is:" << temp->data << endl;

count++;

}

temp = temp->next;

} while (temp != NULL);

if (count == 0)

{

cout << "value not found:" << endl;

}

}

int main()

{

node\* head;

node\* one = NULL;

node\* two = NULL;

node\* three = NULL;

node\* four = NULL;

one = new node();

two = new node();

three = new node();

four = new node();

one->data = 1;

one->next = two;

one->prev = NULL;

two->data = 2;

two->next = three;

two->prev = one;

three->data = 3;

three->next = four;

three->prev = two;

four->data = 4;

four->next = NULL;

four->prev = three;

head = one;

cout << "Linklist before Searching :" << endl;

traverse(&head);

cout << " SEARCHING " << endl;

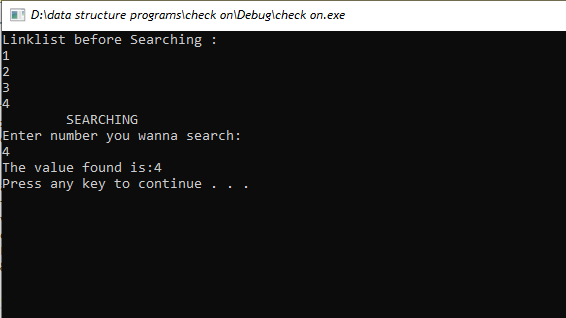
searching(&head);

traverse(&head);

system("pause");

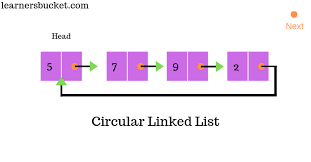
return 0;

}



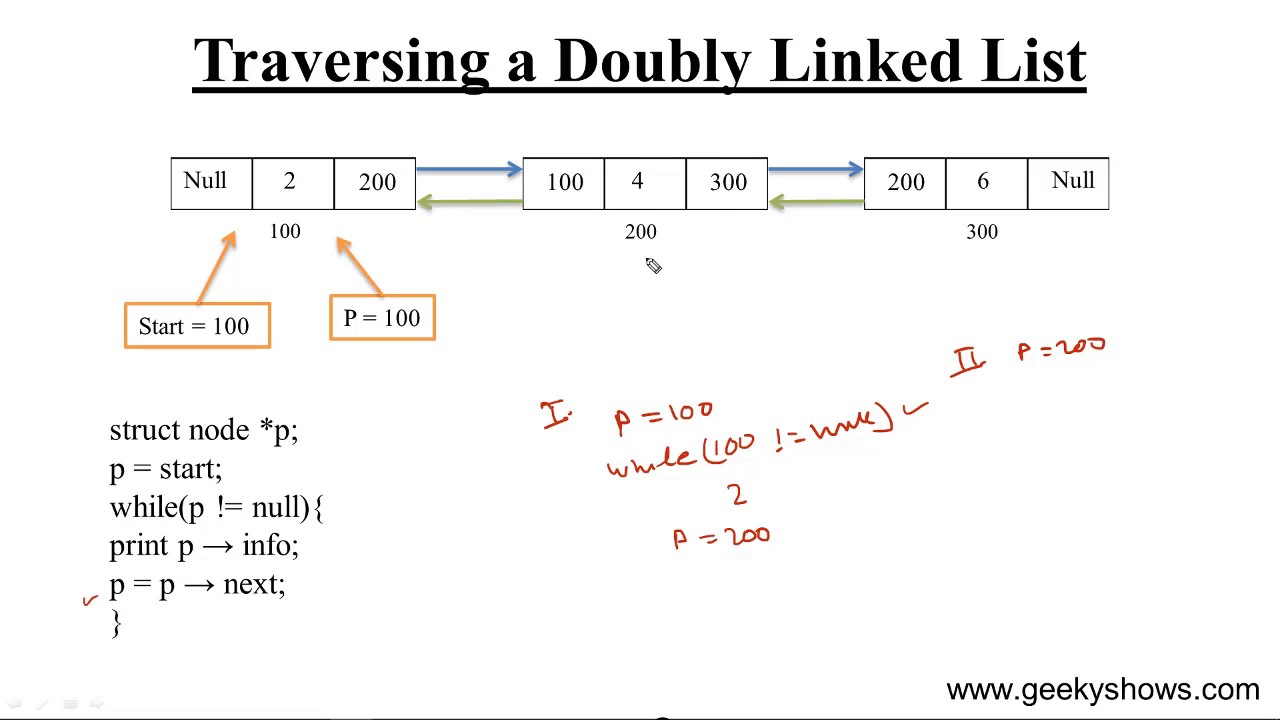
1. **CIRCULAR LINKLIST:**

Single Circular Linked List is a type of data structure in which each node has a pointer to the next node, forming a circular chain. The last node points back to the first node, forming a loop. This structure is useful for applications where a list or queue needs to be maintained and traversed in a circular manner.



1. **TRAVERSING:**

Traversal in a single circular linked list is the process of iterating through all the elements of a list. It involves starting at a particular node, and then visiting each node in sequence until the starting node is reached again. During the traversal, each node's data can be accessed and/or updated.



Code:

#include "stdafx.h"

#include<iostream>

using namespace std;

class node {

public:

int data;

node\* next;

};

void traverse(node\*\* head)

{

node\* temp = \*head;

do {

cout << temp->data << endl;

temp = temp->next;

} while (temp != \*head );

}

int main()

{

node\* head;

node\* one = NULL;

node\* two = NULL;

node\* three = NULL;

node\* four = NULL;

one = new node();

two = new node();

three = new node();

four = new node();

one->data = 1;

one->next = two;

two->data = 2;

two->next = three;

three->data = 3;

three->next = four;

four->data = 4;

four->next = one;

head = one;

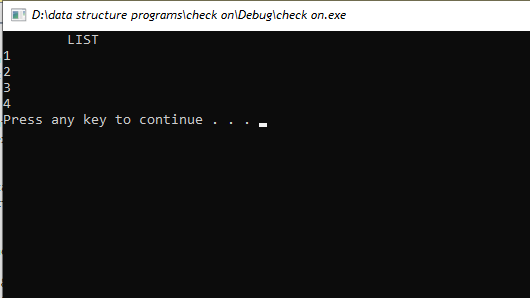
cout << " LIST " << endl;

traverse(&head);

system("pause");

return 0;

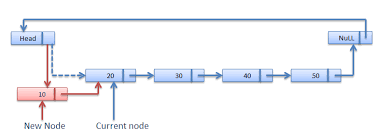
}



1. **INSERTION:**

* **INSERTION\_AT\_START:**

Insertion at the start of a single circular linked list involves creating a new node and linking it to the head of the list. The new node becomes the head and the last node's next pointer is set to point to the new head. This insertion operation takes constant time complexity, O(1).



Code:

#include "stdafx.h"

#include<iostream>

using namespace std;

class node {

public:

int data;

node\* next;

};

void traverse(node\*\* head)

{

node\* temp = \*head;

do {

cout << temp->data << endl;

temp = temp->next;

} while (temp != \*head );

}

void insert\_at\_start(node\*\* head)

{

node\* temp = \*head;

node\* newnode = NULL;

newnode = new node();

newnode->data = 7;

while (temp->next != \*head)

{

temp = temp->next;

}

temp->next = newnode;

newnode->next = \*head;

\*head = newnode;

newnode->data = 0;

}

int main()

{

node\* head;

node\* one = NULL;

node\* two = NULL;

node\* three = NULL;

node\* four = NULL;

one = new node();

two = new node();

three = new node();

four = new node();

one->data = 1;

one->next = two;

two->data = 2;

two->next = three;

three->data = 3;

three->next = four;

four->data = 4;

four->next = one;

head = one;

cout << " LIST BEFORE INSERTION " << endl;

traverse(&head);

cout << " INSERTION AT START " << endl;

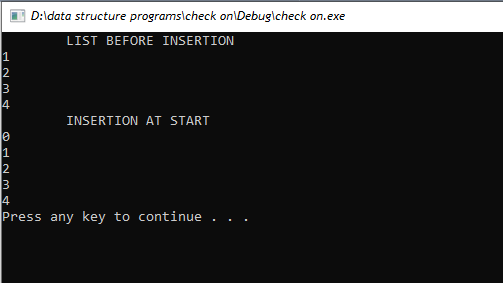
insert\_at\_start(&head);

traverse(&head);

system("pause");

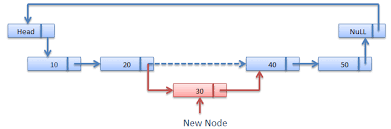
return 0;

}



* **INSERTION\_AT\_MIDDLE:**

Insertion at the middle in a single circular linked list means inserting a node at a given position in the list. The new node is inserted between two nodes of the list, such that the next pointer of the previous node points to the new node and the previous pointer of the next node points to the new node.



#include "stdafx.h"

#include<iostream>

using namespace std;

class node {

public:

int data;

node\* next;

};

void traverse(node\*\* head)

{

node\* temp = \*head;

do {

cout << temp->data << endl;

temp = temp->next;

} while (temp != \*head );

}

void insert\_at\_middle(node\*\* head, int pos)

{

node\* temp = \*head;

node\* midnode = NULL;

midnode = new node();

for (int i = 1; i<pos - 1; i++)

{

temp = temp->next;

}

midnode->data = 9;

midnode->next = temp->next;;

temp->next = midnode;

}

int main()

{

node\* head;

node\* one = NULL;

node\* two = NULL;

node\* three = NULL;

node\* four = NULL;

one = new node();

two = new node();

three = new node();

four = new node();

one->data = 1;

one->next = two;

two->data = 2;

two->next = three;

three->data = 3;

three->next = four;

four->data = 4;

four->next = one;

head = one;

cout << " LIST BEFORE INSERTION " << endl;

traverse(&head);

int pos;

cout << "Enter position u want to insert node " << endl;

cin >> pos;

cout << " INSERTION AT MIDDLE " << endl;

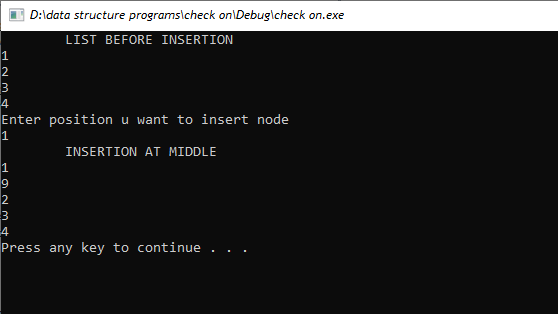
insert\_at\_middle(&head, pos);

traverse(&head);

system("pause");

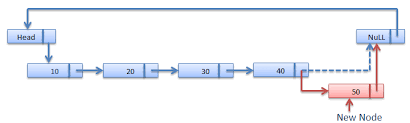
return 0;

}



* **INSERTION\_AT\_END**

Insertion at the end in a single circular linked list is the process of adding a new node to the end of the list, where the new node points back to the first node in the list. This process is useful for adding elements to a list without traversing the entire list.



#include "stdafx.h"

#include<iostream>

using namespace std;

class node {

public:

int data;

node\* next;

};

void traverse(node\*\* head)

{

node\* temp = \*head;

do {

cout << temp->data << endl;

temp = temp->next;

} while (temp != \*head );

}

void insert\_at\_end(node\*\* head)

{

node\* temp = \*head;

node\* endnode = NULL;

endnode = new node();

while (temp->next != \*head)

{

temp = temp->next;

}

endnode->data = 7;

endnode->next = \*head;

temp->next = endnode;

}

int main()

{

node\* head;

node\* one = NULL;

node\* two = NULL;

node\* three = NULL;

node\* four = NULL;

one = new node();

two = new node();

three = new node();

four = new node();

one->data = 1;

one->next = two;

two->data = 2;

two->next = three;

three->data = 3;

three->next = four;

four->data = 4;

four->next = one;

head = one;

cout << " LIST BEFORE INSERTION " << endl;

traverse(&head);

cout << " INSERTION AT END " << endl;

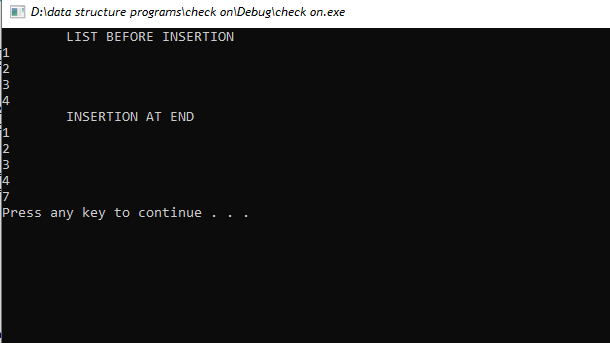
insert\_at\_end(&head);

traverse(&head);

system("pause");

return 0;

}



1. **UPDATION :**

Updation in single circular linklist involves modifying the data associated with the node or replacing a node with a new node. It can also involve inserting a new node after a particular node, deleting a node and rearranging the nodes by changing the links between them.

* **UPDATION\_AT\_START:**

Updation in a single circular linked list is a process of updating existing nodes or adding new nodes to the list. It involves traversing the list to locate the desired node, modifying or inserting the new node, and then linking it to the other nodes in the list. This ensures that the list remains circular and that all nodes are connected together.

#include "stdafx.h"

#include<iostream>

using namespace std;

class node {

public:

int data;

node\* next;

};

void traverse(node\*\* head)

{

node\* temp = \*head;

do {

cout << temp->data << endl;

temp = temp->next;

} while (temp != \*head );

}

void updation\_at\_start(node \*\*head, int x)

{

node \*temp = \*head;

temp->data = x;

}

int main()

{

node\* head;

node\* one = NULL;

node\* two = NULL;

node\* three = NULL;

node\* four = NULL;

one = new node();

two = new node();

three = new node();

four = new node();

one->data = 1;

one->next = two;

two->data = 2;

two->next = three;

three->data = 3;

three->next = four;

four->data = 4;

four->next = one;

head = one;

cout << " LIST BEFORE UPDATIOB " << endl;

traverse(&head);

cout << " UPDATION AT START " << endl;

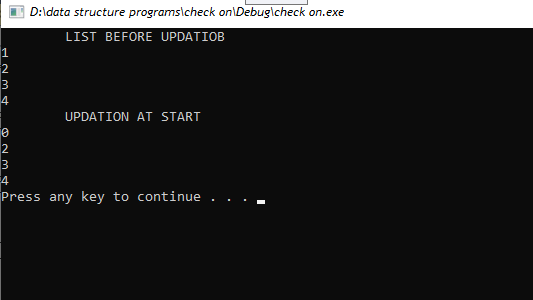
updation\_at\_start(&head, 0);

traverse(&head);

system("pause");

return 0;

}



* **UPDATION\_AT\_MIDDLE:**

Updation at mid in single circular linked list is the process of modifying a node in a single circular linked list which is present in the middle of the list. It involves changing the data stored in the node, as well as updating the link pointers to shift the node to a different location in the list.

#include "stdafx.h"

#include<iostream>

using namespace std;

class node {

public:

int data;

node\* next;

};

void traverse(node\*\* head)

{

node\* temp = \*head;

do {

cout << temp->data << endl;

temp = temp->next;

} while (temp != \*head );

}

void updation\_at\_middle(node \*\*head, int x)

{

int pos;

cout << "Enter the postion u want to update " << endl;

cin >> pos;

node \*temp = \*head;

for (int i = 1; i < pos; i++)

{

temp = temp->next;

}

temp->data = x;

}

int main()

{

node\* head;

node\* one = NULL;

node\* two = NULL;

node\* three = NULL;

node\* four = NULL;

one = new node();

two = new node();

three = new node();

four = new node();

one->data = 1;

one->next = two;

two->data = 2;

two->next = three;

three->data = 3;

three->next = four;

four->data = 4;

four->next = one;

head = one;

cout << " LIST BEFORE UPDATION " << endl;

traverse(&head);

cout << " UPDATION AT MIDDLE " << endl;

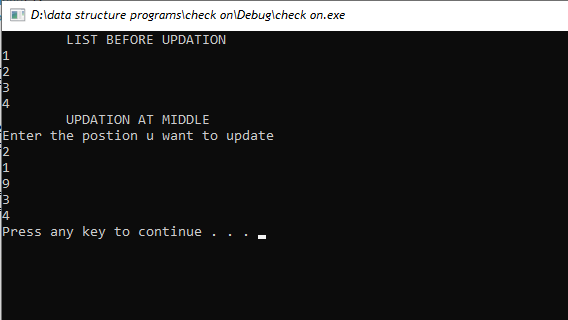
updation\_at\_middle(&head, 9);

traverse(&head);

system("pause");

return 0;

}



* **UPDATION\_AT\_END:**

An updation at the end of a single circular linked list is a process in which the last node of the list is updated with new data or a new node is added after the last node of the list. This process helps in maintaining the data structure of the list and ensuring that the list remains circular.

#include "stdafx.h"

#include<iostream>

using namespace std;

class node {

public:

int data;

node\* next;

};

void traverse(node\*\* head)

{

node\* temp = \*head;

do {

cout << temp->data << endl;

temp = temp->next;

} while (temp != \*head );

}

void updation\_at\_end(node \*\*head, int x)

{

node \*temp = \*head;

while (temp->next != \*head)

{

temp = temp->next;

}

temp->data = x;

}

int main()

{

node\* head;

node\* one = NULL;

node\* two = NULL;

node\* three = NULL;

node\* four = NULL;

one = new node();

two = new node();

three = new node();

four = new node();

one->data = 1;

one->next = two;

two->data = 2;

two->next = three;

three->data = 3;

three->next = four;

four->data = 4;

four->next = one;

head = one;

cout << " LIST BEFORE UPDATION " << endl;

traverse(&head);

cout << " UPDATION AT END " << endl;

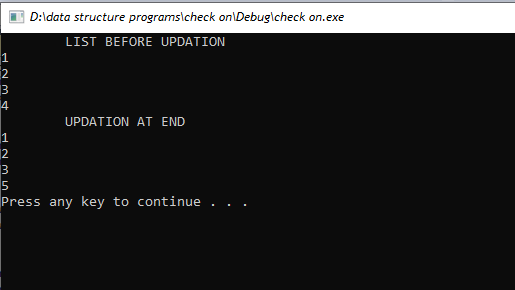
updation\_at\_end(&head, 5);

traverse(&head);

system("pause");

return 0;

}

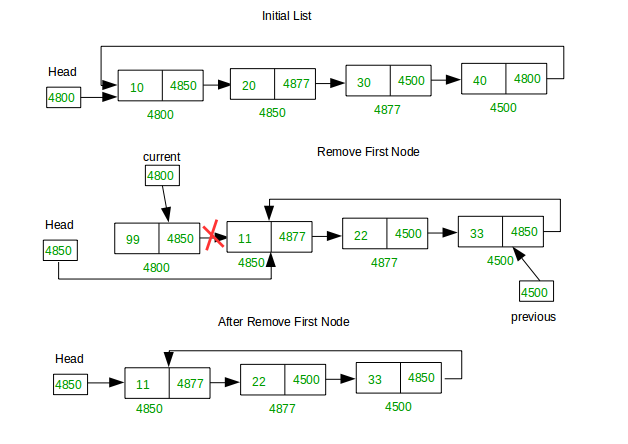


1. **DELETION :**

Deletion in single circular linked list is the process of deleting a node from the list. It involves changing the links of the nodes before and after the one to be deleted such that the list remains circular. It is used in applications such as memory management, garbage collection, and data structure manipulation.

* **DELETION\_AT\_START:**

Deletion in single circular linked list is the process of removing a node from the linked list. It involves updating the head and tail pointers to ensure that the linked list remains circular, and setting the next pointer of the node before the deleted node to point to the node after the deleted one.



#include "stdafx.h"

#include<iostream>

using namespace std;

class node {

public:

int data;

node\* next;

};

void traverse(node\*\* head)

{

node\* temp = \*head;

do {

cout << temp->data << endl;

temp = temp->next;

} while (temp != \*head );

}

void deletion\_at\_start(node \*\*head)

{

node \*temp=\*head;

node \*temp1 = \*head;

while (temp->next != \*head)

{

temp=temp->next;

}

temp->next = temp1->next;

\*head = temp1->next;

delete(temp1);

}

int main()

{

node\* head;

node\* one = NULL;

node\* two = NULL;

node\* three = NULL;

node\* four = NULL;

one = new node();

two = new node();

three = new node();

four = new node();

one->data = 1;

one->next = two;

two->data = 2;

two->next = three;

three->data = 3;

three->next = four;

four->data = 4;

four->next = one;

head = one;

cout << " LIST BEFORE DELETION " << endl;

traverse(&head);

cout << " DELETION AT START " << endl;

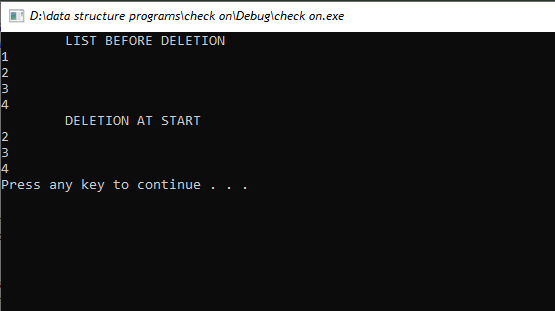
deletion\_at\_start(&head);

traverse(&head);

system("pause");

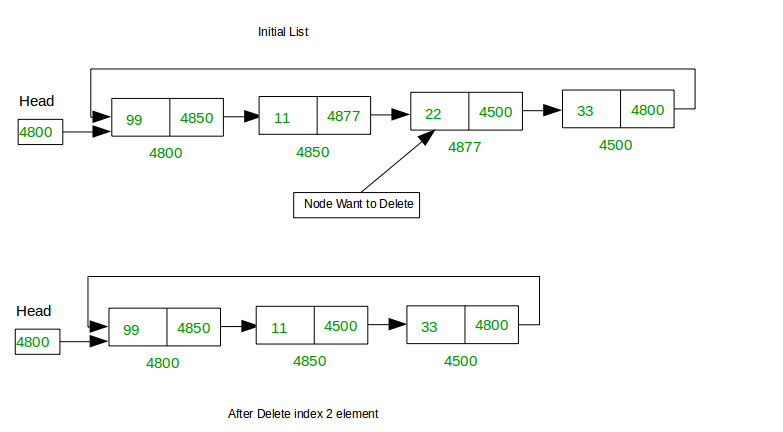
return 0;

}



* **DELETION\_AT\_MIDDLE:**

Deletion at the middle of a single circular link list involves removing an element from the list, while maintaining the link between the previous and the next element. It requires a pointer to the node which is to be deleted and the head of the list, so that the link to the next element can be adjusted.



#include "stdafx.h"

#include<iostream>

using namespace std;

class node {

public:

int data;

node\* next;

};

void traverse(node\*\* head)

{

node\* temp = \*head;

do {

cout << temp->data << endl;

temp = temp->next;

} while (temp != \*head );

}

void deletion\_at\_middle(node \*\*head)

{

int pos;

cout << "Enter the position u want to delete" << endl;

cin >> pos;

node \*temp1 = NULL;

node \*temp = \*head;

for (int i = 1; i < pos; i++)

{

temp1 = temp;

temp = temp->next;

}

temp1->next = temp->next;

delete(temp);

}

int main()

{

node\* head;

node\* one = NULL;

node\* two = NULL;

node\* three = NULL;

node\* four = NULL;

one = new node();

two = new node();

three = new node();

four = new node();

one->data = 1;

one->next = two;

two->data = 2;

two->next = three;

three->data = 3;

three->next = four;

four->data = 4;

four->next = one;

head = one;

cout << " LIST BEFORE DELETION " << endl;

traverse(&head);

cout << " DELETION AT MIDDLE " << endl;

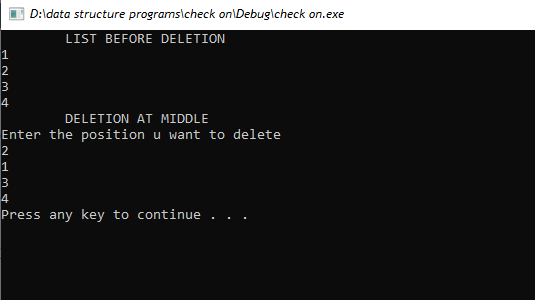
deletion\_at\_middle(&head);

traverse(&head);

system("pause");

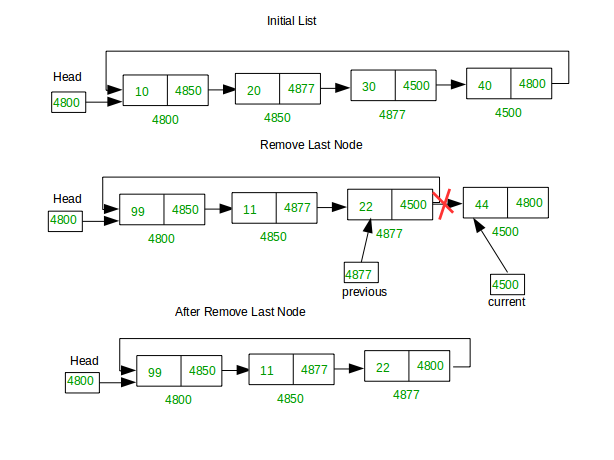
return 0;

}



* **DELETION\_AT\_END:**

In a single circular linked list, deletion is the process of removing a node from the list and reconnecting the remaining nodes. It involves changing the pointers of the nodes to the left and right of the node to be deleted, so that the list remains circular.



#include "stdafx.h"

#include<iostream>

using namespace std;

class node {

public:

int data;

node\* next;

};

void traverse(node\*\* head)

{

node\* temp = \*head;

do {

cout << temp->data << endl;

temp = temp->next;

} while (temp != \*head );

}

void deletion\_at\_end(node \*\*head)

{

node \*temp = \*head;

node \*temp1=\*head;

while (temp->next != \*head)

{

temp1 = temp;

temp = temp->next;

}

temp1->next = \*head;

delete(temp);

}

int main()

{

node\* head;

node\* one = NULL;

node\* two = NULL;

node\* three = NULL;

node\* four = NULL;

one = new node();

two = new node();

three = new node();

four = new node();

one->data = 1;

one->next = two;

two->data = 2;

two->next = three;

three->data = 3;

three->next = four;

four->data = 4;

four->next = one;

head = one;

cout << " LIST BEFORE DELETION " << endl;

traverse(&head);

cout << " DELETION AT END " << endl;

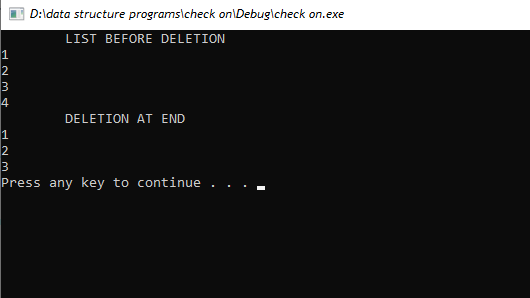
deletion\_at\_end(&head);

traverse(&head);

system("pause");

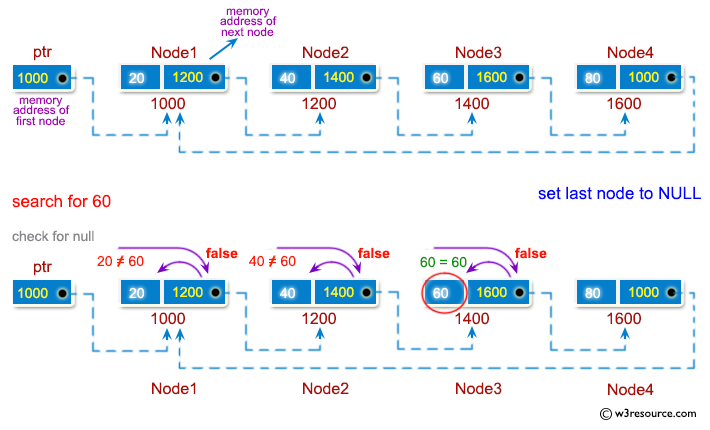
return 0;

}



1. **SEARCHING:**

Searching in a single circular linklist is a process of traversing a list of elements to find a specific element. It is done by starting at one node and following the link of each node until the element is found or the end of the list is reached. The list is considered circular because the last node links back to the first node, thus allowing the search to start from any node in the list.



#include "stdafx.h"

#include<iostream>

using namespace std;

class node {

public:

int data;

node\* next;

};

void traverse(node\*\* head)

{

node\* temp = \*head;

do {

cout << temp->data << endl;

temp = temp->next;

} while (temp != \*head );

}

void searching(node \*\*head)

{

int a, count = 0;

cout << "Enter number you wanna search:" << endl;

cin >> a;

node\*temp = \*head;

do{

if (temp->data == a)

{

cout << "The value found is:" << temp->data << endl;

count++;

}

temp = temp->next;

} while (temp != \*head);

if (count == 0)

{

cout << "value not found:" << endl;

}

}

int main()

{

node\* head;

node\* one = NULL;

node\* two = NULL;

node\* three = NULL;

node\* four = NULL;

one = new node();

two = new node();

three = new node();

four = new node();

one->data = 1;

one->next = two;

two->data = 2;

two->next = three;

three->data = 3;

three->next = four;

four->data = 4;

four->next = one;

head = one;

cout << " LIST " << endl;

traverse(&head);

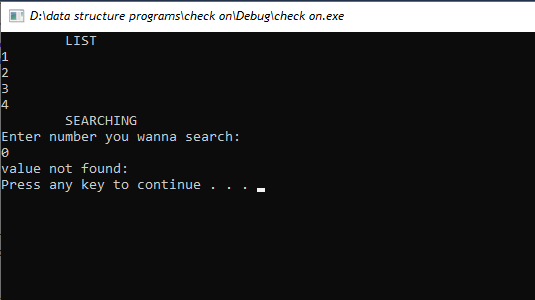
cout << " SEARCHING " << endl;

searching(&head);

system("pause");

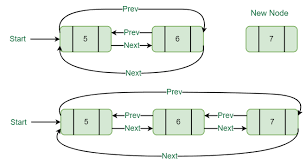
return 0;

}



1. **CIRCULAR DOUBLE LINKLIST:**

Double circular linked list is a variation of the circular linked list in which each node has two links, one pointing to the next node and one pointing to the previous node. This allows for traversal in both directions. It is useful for applications such as maintaining a list of recently accessed items, allowing for both forward and backward movement within the list.



1. **TRAVERSING:**

Traversal in double circular linked list is the process of visiting each node of the linked list in a sequential manner, starting from the head node and ending at the tail node. The process is repeated until all the nodes have been visited. It is similar to regular linked list traversal but has the advantage of being able to traverse back to the beginning of the list.

#include "stdafx.h"

#include<iostream>

using namespace std;

class node {

public:

int data;

node\* next;

node\* prev;

};

void traverse(node\*\* head)

{

node\* temp = \*head;

do

{

cout << temp->data << endl;

temp = temp->next;

} while (temp != \*head);

}

int main()

{

node\* head;

node\* one = NULL;

node\* two = NULL;

node\* three = NULL;

node\* four = NULL;

one = new node();

two = new node();

three = new node();

four = new node();

one->data = 1;

one->next = two;

one->prev = four;

two->data = 2;

two->next = three;

two->prev = one;

three->data = 3;

three->next = four;

three->prev = two;

four->data = 4;

four->next = one;

four->prev = three;

head = one;

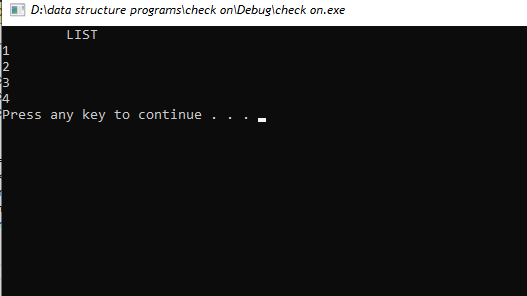
cout << " LIST " << endl;

traverse(&head);

system("pause");

return 0;

}

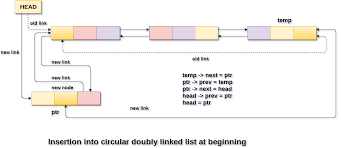


1. **INSERTION:**

Insertion in a double circular linked list is the process of creating a new node and adding it to the list. The new node is added between two existing nodes so that it becomes part of the list. Insertion is an important operation in double circular linked lists, as it allows for the addition of a new element without disturbing the existing elements. Insertion can be done at the beginning, middle, or end of the list.

* **INSERTION\_AT\_START:**

Insertion at the start of a double circular linked list involves creating a new node and pointing the last node's next pointer to the new node and then pointing the new node's next pointer back to the first node of the list. This process is also known as the head insertion, as the new node becomes the head of the list.



#include "stdafx.h"

#include<iostream>

using namespace std;

class node {

public:

int data;

node\* next;

node\* prev;

};

void traverse(node\*\* head)

{

node\* temp = \*head;

do

{

cout << temp->data << endl;

temp = temp->next;

} while (temp != \*head);

}

void insert\_at\_start(node\*\* head)

{

node\* temp = \*head;

node\* newnode = NULL;

newnode = new node();

newnode->data = 7;

temp->prev->next = newnode;

temp->prev = newnode;

newnode->next = \*head;

\*head = newnode;

}

int main()

{

node\* head;

node\* one = NULL;

node\* two = NULL;

node\* three = NULL;

node\* four = NULL;

one = new node();

two = new node();

three = new node();

four = new node();

one->data = 1;

one->next = two;

one->prev = four;

two->data = 2;

two->next = three;

two->prev = one;

three->data = 3;

three->next = four;

three->prev = two;

four->data = 4;

four->next = one;

four->prev = three;

head = one;

cout << " LIST BEFORE INSERTION " << endl;

traverse(&head);

cout << " INSERTION AT START " << endl;

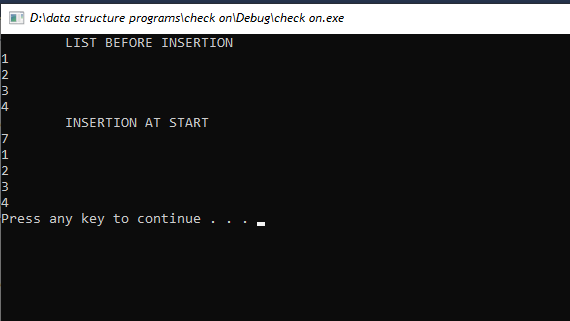
insert\_at\_start(&head);

traverse(&head);

system("pause");

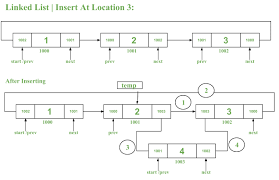
return 0;

}



* **INSERTION\_AT\_MIDDLE:**

Insertion at the middle of a double circular linked list is a process of adding a new node at the middle of existing nodes in a double circular linked list. It is a process of shifting the existing nodes and connecting the new node with the existing nodes.



#include "stdafx.h"

#include<iostream>

using namespace std;

class node {

public:

int data;

node\* next;

node\* prev;

};

void traverse(node\*\* head)

{

node\* temp = \*head;

do

{

cout << temp->data << endl;

temp = temp->next;

} while (temp != \*head);

}

void insert\_at\_middle(node\*\* head, int pos)

{

node\* temp = \*head;

node\* midnode = NULL;

midnode = new node();

for (int i = 1; i<pos - 1; i++)

{

temp = temp->next;

}

midnode->data = 0;

midnode->next = temp->next;

temp->next->prev = midnode;

temp->next = midnode;

midnode->prev = temp;

}

int main()

{

node\* head;

node\* one = NULL;

node\* two = NULL;

node\* three = NULL;

node\* four = NULL;

one = new node();

two = new node();

three = new node();

four = new node();

one->data = 1;

one->next = two;

one->prev = four;

two->data = 2;

two->next = three;

two->prev = one;

three->data = 3;

three->next = four;

three->prev = two;

four->data = 4;

four->next = one;

four->prev = three;

head = one;

cout << " LIST BEFORE INSERTION " << endl;

traverse(&head);

int pos;

cout << "Enter position u want to insert node " << endl;

cin >> pos;

cout << " INSERTION AT MIDDLE " << endl;

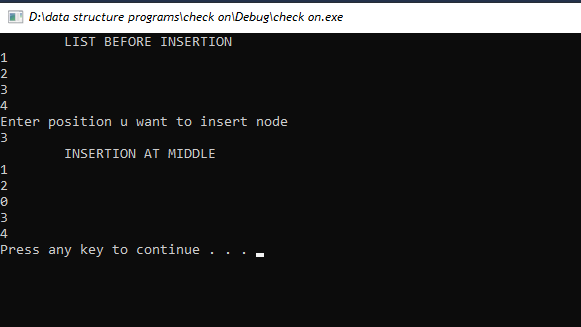
insert\_at\_middle(&head, pos);

traverse(&head);

system("pause");

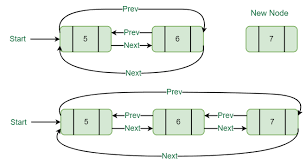
return 0;

}



* **INSERTION\_AT\_END:**

Insertion at the end in double circular linked list is a data structure operation in which a new node is added at the end of an existing double circular linked list. It involves creating a new node, linking the new node to the last node in the list and setting the next pointer of the last node to the new node. The new node's previous pointer is set to the last node and its next pointer is set to the head of the list, thus creating a double circular linked list.



#include "stdafx.h"

#include<iostream>

using namespace std;

class node {

public:

int data;

node\* next;

node\* prev;

};

void traverse(node\*\* head)

{

node\* temp = \*head;

do

{

cout << temp->data << endl;

temp = temp->next;

} while (temp != \*head);

}

void insert\_at\_end(node\*\* head)

{

node\* temp = \*head;

node\* endnode = NULL;

endnode = new node();

while (temp->next != \*head)

{

temp = temp->next;

}

endnode->data = 9;

endnode->next = \*head;

endnode->prev = temp;

temp->next = endnode;

}

int main()

{

node\* head;

node\* one = NULL;

node\* two = NULL;

node\* three = NULL;

node\* four = NULL;

one = new node();

two = new node();

three = new node();

four = new node();

one->data = 1;

one->next = two;

one->prev = four;

two->data = 2;

two->next = three;

two->prev = one;

three->data = 3;

three->next = four;

three->prev = two;

four->data = 4;

four->next = one;

four->prev = three;

head = one;

cout << " LIST BEFORE INSERTION " << endl;

traverse(&head);

cout << " INSERTION AT END " << endl;

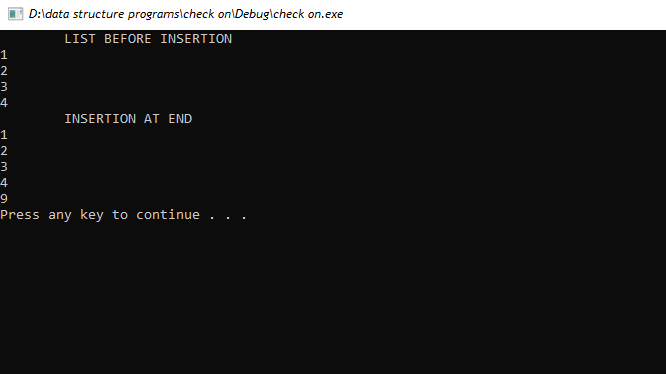
insert\_at\_end(&head);

traverse(&head);

system("pause");

return 0;

}



1. **UPDATION:**

Updation of element in a double circular linked list can be done by first traversing the list to search for the node which contains the element to be updated, then updating the data stored in the node after the node is found. After the updation is done, the list is traversed again to make sure that the updated elements are in the right position.

* **UPDATION\_AT\_START:**

Updation at the start of a double circular link list involves adding a new node at the start of the list. The new node is added to the list by changing the pointers of the head node and the last node. The newly added node becomes the new head node of the list.

#include "stdafx.h"

#include<iostream>

using namespace std;

class node {

public:

int data;

node\* next;

node\* prev;

};

void traverse(node\*\* head)

{

node\* temp = \*head;

do

{

cout << temp->data << endl;

temp = temp->next;

} while (temp != \*head);

}

void updation\_at\_start(node \*\*head, int x)

{

node \*temp = \*head;

temp->data = x;

}

int main()

{

node\* head;

node\* one = NULL;

node\* two = NULL;

node\* three = NULL;

node\* four = NULL;

one = new node();

two = new node();

three = new node();

four = new node();

one->data = 1;

one->next = two;

one->prev = four;

two->data = 2;

two->next = three;

two->prev = one;

three->data = 3;

three->next = four;

three->prev = two;

four->data = 4;

four->next = one;

four->prev = three;

head = one;

cout << " LIST BEFORE UPDATION " << endl;

traverse(&head);

cout << " UPDATION AT START " << endl;

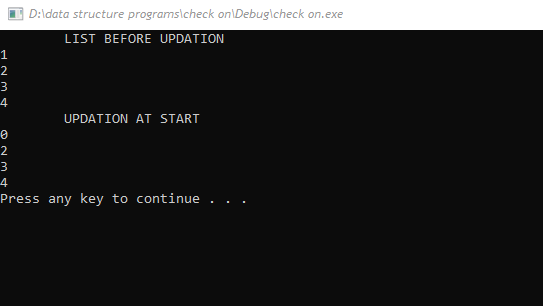
updation\_at\_start(&head, 0);

traverse(&head);

system("pause");

return 0;

}



* **UPDATION\_AT\_MIDDLE:**

An updation at the middle of a double circular linked list involves updating a node in the middle of the list without compromising the circular links of the list. This can be done by first traversing the list to find the desired node and then updating the data of the node as per requirement. Finally, the node's pointer links should be updated to maintain the integrity of the list.

#include "stdafx.h"

#include<iostream>

using namespace std;

class node {

public:

int data;

node\* next;

node\* prev;

};

void traverse(node\*\* head)

{

node\* temp = \*head;

do

{

cout << temp->data << endl;

temp = temp->next;

} while (temp != \*head);

}

void updation\_at\_middle(node \*\*head, int x)

{

int pos;

cout << "Enter the postion u want to update " << endl;

cin >> pos;

node \*temp = \*head;

for (int i = 1; i < pos; i++)

{

temp = temp->next;

}

temp->data = x;

}

int main()

{

node\* head;

node\* one = NULL;

node\* two = NULL;

node\* three = NULL;

node\* four = NULL;

one = new node();

two = new node();

three = new node();

four = new node();

one->data = 1;

one->next = two;

one->prev = four;

two->data = 2;

two->next = three;

two->prev = one;

three->data = 3;

three->next = four;

three->prev = two;

four->data = 4;

four->next = one;

four->prev = three;

head = one;

cout << " LIST BEFORE UPDATION " << endl;

traverse(&head);

cout << " UPDATION AT MIDDLE " << endl;

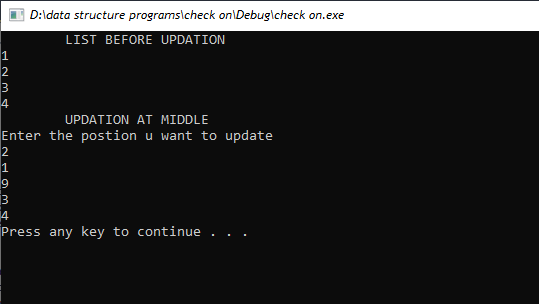
updation\_at\_middle(&head, 9);

traverse(&head);

system("pause");

return 0;

}



* **UPDATION\_AT\_END:**

Updation in a double circular linked list is performed when an element is added, deleted, or modified. This can be done by traversing the list and making the necessary changes to the links. Updating a double circular linked list is an efficient process as it requires minimal changes to t he structure of the list.

#include "stdafx.h"

#include<iostream>

using namespace std;

class node {

public:

int data;

node\* next;

node\* prev;

};

void traverse(node\*\* head)

{

node\* temp = \*head;

do

{

cout << temp->data << endl;

temp = temp->next;

} while (temp != \*head);

}

void updation\_at\_end(node \*\*head, int x)

{

node \*temp = \*head;

while (temp->next != \*head)

{

temp = temp->next;

}

temp->data = x;

}

int main()

{

node\* head;

node\* one = NULL;

node\* two = NULL;

node\* three = NULL;

node\* four = NULL;

one = new node();

two = new node();

three = new node();

four = new node();

one->data = 1;

one->next = two;

one->prev = four;

two->data = 2;

two->next = three;

two->prev = one;

three->data = 3;

three->next = four;

three->prev = two;

four->data = 4;

four->next = one;

four->prev = three;

head = one;

cout << " LIST BEFORE UPDATION " << endl;

traverse(&head);

cout << " UPDATION AT END " << endl;

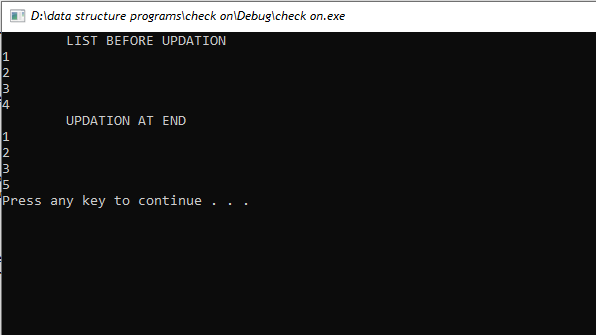
updation\_at\_end(&head, 5);

traverse(&head);

system("pause");

return 0;

}

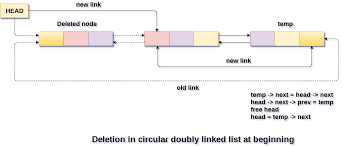


1. **DELETION:**

Deletion of an element in a double circular linked list involves updating the pointers of the previous and next nodes of the node to be deleted to point to each other, thus removing the node from the list. Additionally, the start and end pointers of the list may need to be updated if the node being deleted is the start or end node of the list.

* **DELETION\_AT\_START:**

Deletion of an element at the start of the double circular linked list is done by making the head pointer point to the next element of the list and making the prev pointer of the next element point to the last element of the list. The last element's next pointer is then made to point to the new head node. This process is done in constant time.



#include "stdafx.h"

#include<iostream>

using namespace std;

class node {

public:

int data;

node\* next;

node\* prev;

};

void traverse(node\*\* head)

{

node\* temp = \*head;

do

{

cout << temp->data << endl;

temp = temp->next;

} while (temp != \*head);

}

void deletion\_at\_start(node \*\*head)

{

node \*temp = \*head;

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

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

\*head = temp->next;

delete(temp);

}

int main()

{

node\* head;

node\* one = NULL;

node\* two = NULL;

node\* three = NULL;

node\* four = NULL;

one = new node();

two = new node();

three = new node();

four = new node();

one->data = 1;

one->next = two;

one->prev = four;

two->data = 2;

two->next = three;

two->prev = one;

three->data = 3;

three->next = four;

three->prev = two;

four->data = 4;

four->next = one;

four->prev = three;

head = one;

cout << " LIST BEFORE DELETION " << endl;

traverse(&head);

cout << " DELETION AT START " << endl;

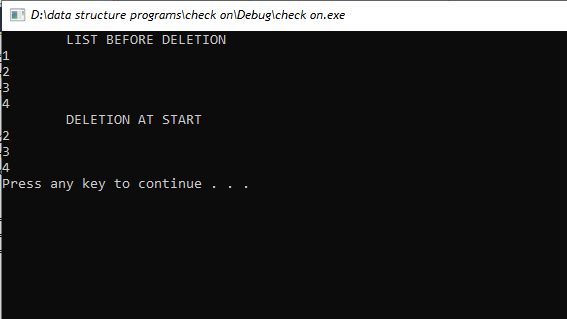
deletion\_at\_start(&head);

traverse(&head);

system("pause");

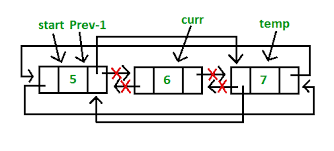
return 0;

}



* **DELETION\_AT\_MIDDLE:**

Deleting an element at the middle of a double circular linked list involves finding the node before the one to be deleted, pointing its next pointer to the next node of the one to be deleted, and then pointing the previous pointer of the next node to the node before the one to be deleted. Finally, the node to be deleted is removed from the list.



#include "stdafx.h"

#include<iostream>

using namespace std;

class node {

public:

int data;

node\* next;

node\* prev;

};

void traverse(node\*\* head)

{

node\* temp = \*head;

do

{

cout << temp->data << endl;

temp = temp->next;

} while (temp != \*head);

}

void deletion\_at\_middle(node \*\*head)

{

int pos;

cout << "Enter the position u want to delete" << endl;

cin >> pos;

node \*temp = \*head;

for (int i = 1; i < pos; i++)

{

temp = temp->next;

}

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

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

delete(temp);

}

int main()

{

node\* head;

node\* one = NULL;

node\* two = NULL;

node\* three = NULL;

node\* four = NULL;

one = new node();

two = new node();

three = new node();

four = new node();

one->data = 1;

one->next = two;

one->prev = four;

two->data = 2;

two->next = three;

two->prev = one;

three->data = 3;

three->next = four;

three->prev = two;

four->data = 4;

four->next = one;

four->prev = three;

head = one;

cout << " LIST BEFORE DELETION " << endl;

traverse(&head);

cout << " DELETION AT MIDDLE " << endl;

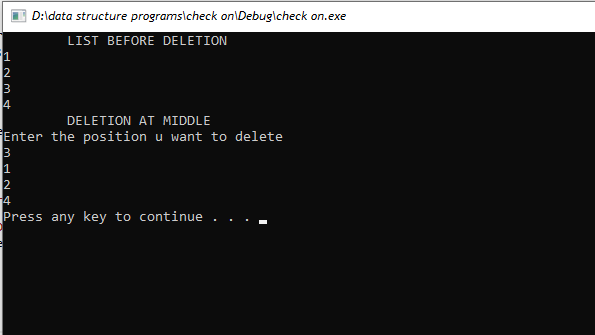
deletion\_at\_middle(&head);

traverse(&head);

system("pause");

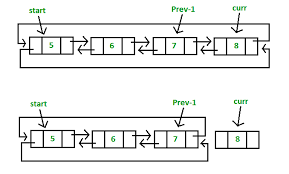
return 0;

}



* **DELETION\_AT\_END:**

Deletion of an element at the end of a double circular linked list is done by making the previous element of the last node point to the head of the list instead of the last node. Then, the last node is removed from the list.



#include "stdafx.h"

#include<iostream>

using namespace std;

class node {

public:

int data;

node\* next;

node\* prev;

};

void traverse(node\*\* head)

{

node\* temp = \*head;

do

{

cout << temp->data << endl;

temp = temp->next;

} while (temp != \*head);

}

void deletion\_at\_end(node \*\*head)

{

node \*temp = \*head;

while (temp->next !=\*head)

{

temp = temp->next;

}

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

delete(temp);

}

int main()

{

node\* head;

node\* one = NULL;

node\* two = NULL;

node\* three = NULL;

node\* four = NULL;

one = new node();

two = new node();

three = new node();

four = new node();

one->data = 1;

one->next = two;

one->prev = four;

two->data = 2;

two->next = three;

two->prev = one;

three->data = 3;

three->next = four;

three->prev = two;

four->data = 4;

four->next = one;

four->prev = three;

head = one;

cout << " LIST BEFORE DELETION " << endl;

traverse(&head);

cout << " DELETION AT END " << endl;

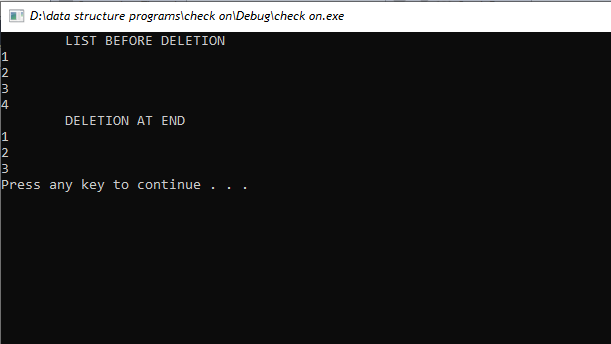
deletion\_at\_end(&head);

traverse(&head);

system("pause");

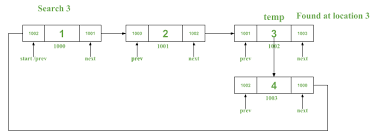
return 0;

}



1. **SEARCHING:**

Double Circular Linked List is a type of linked list in which the last node points to the head node and the head node points to the last node. It is a variation of the circular linked list, where the next and previous pointers of the head node point to the last node and the next and previous pointers of the last node point to the head node. Searching in a double circular linked list can be done in a similar way as searching in a single circular linked list. This involves traversing the list until the desired element is found or the end of the list is reached.



#include "stdafx.h"

#include<iostream>

using namespace std;

class node {

public:

int data;

node\* next;

node\* prev;

};

void traverse(node\*\* head)

{

node\* temp = \*head;

do

{

cout << temp->data << endl;

temp = temp->next;

} while (temp != \*head);

}

void searching(node \*\*head)

{

int a, count = 0;

cout << "Enter number you wanna search:" << endl;

cin >> a;

node\*temp = \*head;

do{

if (temp->data == a)

{

cout << "The value found is:" << temp->data << endl;

count++;

}

temp = temp->next;

} while (temp != \*head);

if (count == 0)

{

cout << "value not found:" << endl;

}

}

int main()

{

node\* head;

node\* one = NULL;

node\* two = NULL;

node\* three = NULL;

node\* four = NULL;

one = new node();

two = new node();

three = new node();

four = new node();

one->data = 1;

one->next = two;

one->prev = four;

two->data = 2;

two->next = three;

two->prev = one;

three->data = 3;

three->next = four;

three->prev = two;

four->data = 4;

four->next = one;

four->prev = three;

head = one;

cout << " LIST " << endl;

traverse(&head);

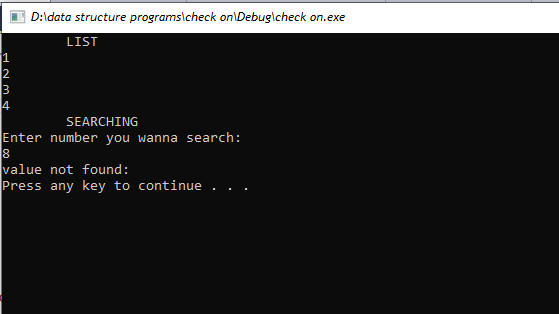
cout << " SEARCHING " << endl;

searching(&head);

system("pause");

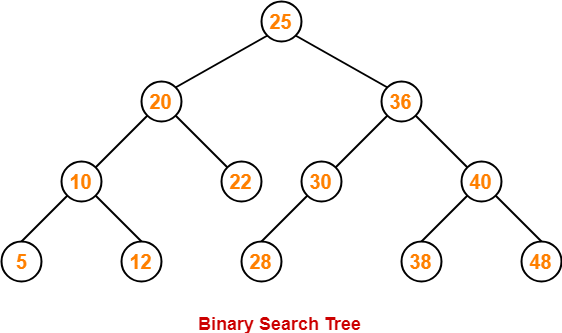
return 0;

}



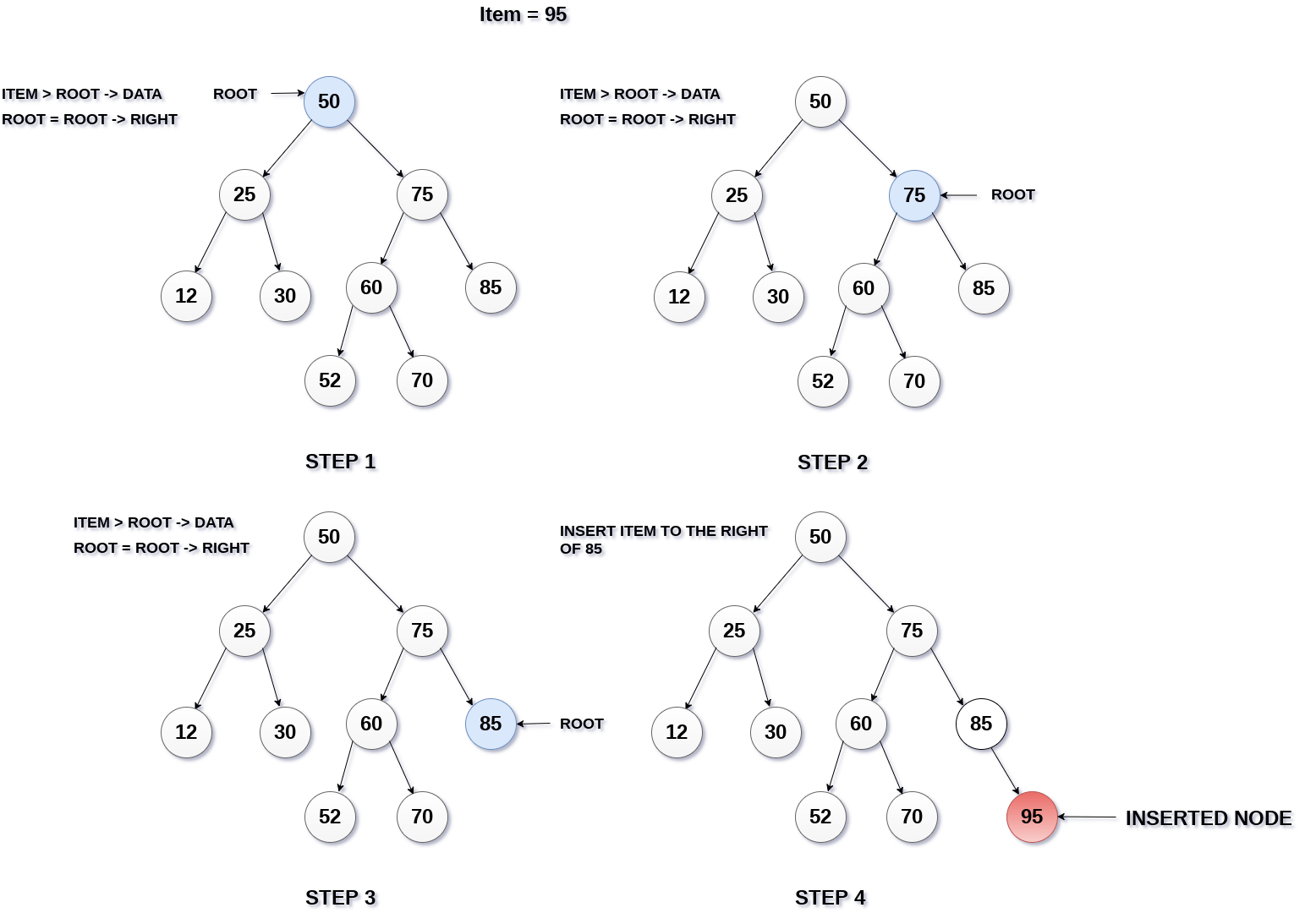
1. **BST TREES:**

Binary search tree is a type of data structure that stores data in a specific way so it can be quickly accessed and sorted. It is a type of tree in which each node has at most two children, left and right, and each node's value is greater than the values in its left sub-tree and smaller than the values in its right sub-tree.



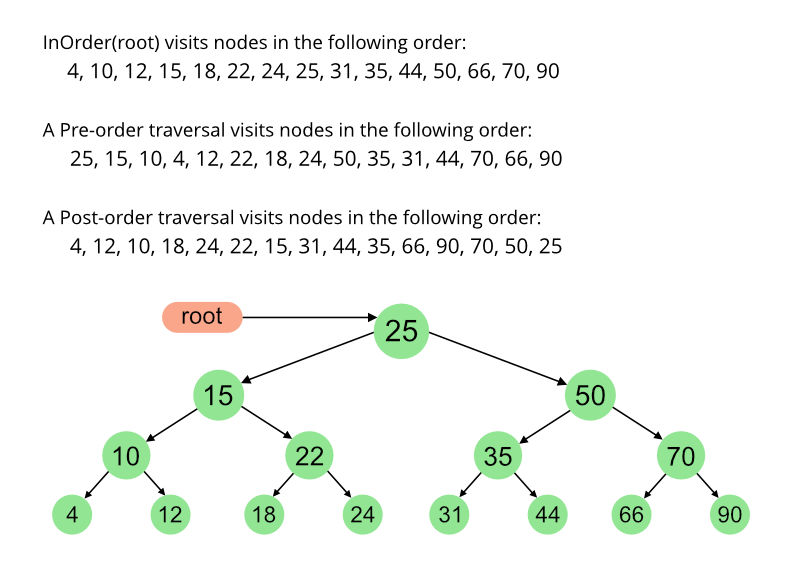
1. **Insertion:**

Insertion in a Binary Search Tree is the process of adding a new node to the tree, ensuring that the tree remains a valid Binary Search Tree. This is accomplished by performing a series of comparisons with the values in the tree, and then deciding where to place the new node.



1. **Traversal:**

Traversal in a Binary Search Tree is the process of visiting each node of the tree at least once. The order in which the nodes are visited determines the type of traversal. Common types of traversals include pre-order, in-order, post-order, and level-order.



* **PREORDER TRAVERSING**

Preorder traversal in a binary search tree is a depth-first traversal method where the root node is visited first, followed by its left subtree, and then its right subtree. The nodes are visited in the order of root, left, right.

#include"stdafx.h"

#include<iostream>

using namespace std;

class BST

{

public:

int data;

BST\*leftchild;

BST\*rightchild;

};

BST\*getvalue(BST\*root\_ref, int value)

{

BST\*newnode = NULL;

newnode = new BST();

newnode->data = value;

newnode->leftchild = NULL;

newnode->rightchild = NULL;

return newnode;

}

BST\*insertion(BST\*root,int value)

{

if (root == NULL)

{

return getvalue(root,value);

}

if (root->data > value)

{

root->leftchild = insertion(root->leftchild, value);

}

else if (root->data < value)

{

root->rightchild = insertion(root->rightchild, value);

}

return root;

}

void preorder(BST \*root)

{

if (root == NULL)

{

return;

}

cout << root->data << endl;

preorder(root->leftchild);

preorder(root->rightchild);

}

void main()

{

BST\*root = NULL;

int value;

root=getvalue(root, 2);

insertion(root, 1);

insertion(root, 4);

insertion(root, 8);

insertion(root, 3);

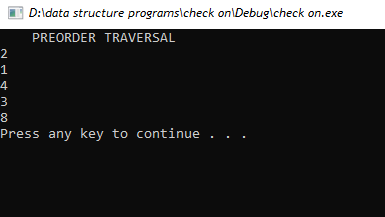
cout << " PREORDER TRAVERSAL " << endl;

preorder(root);

system("pause");

return;

}



* **INORDER TRAVERSING:**

Inorder traversal in a Binary Search Tree is a type of tree traversal where the nodes are visited in the order of left subtree -> root -> right subtree. This is usually used to print the nodes in ascending order.

#include"stdafx.h"

#include<iostream>

using namespace std;

class BST

{

public:

int data;

BST\*leftchild;

BST\*rightchild;

};

BST\*getvalue(BST\*root\_ref, int value)

{

BST\*newnode = NULL;

newnode = new BST();

newnode->data = value;

newnode->leftchild = NULL;

newnode->rightchild = NULL;

return newnode;

}

BST\*insertion(BST\*root,int value)

{

if (root == NULL)

{

return getvalue(root,value);

}

if (root->data > value)

{

root->leftchild = insertion(root->leftchild, value);

}

else if (root->data < value)

{

root->rightchild = insertion(root->rightchild, value);

}

return root;

}

void inorder(BST \*root)

{

if (root == NULL)

{

return;

}

inorder(root->leftchild);

cout << root->data << endl;

inorder(root->rightchild);

}

void main()

{

BST\*root = NULL;

int value;

root=getvalue(root, 2);

insertion(root, 1);

insertion(root, 4);

insertion(root, 8);

insertion(root, 3);

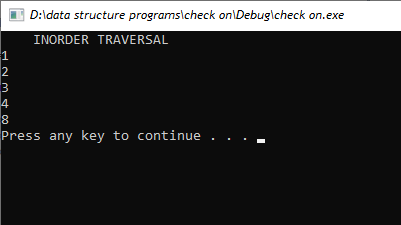
cout << " INORDER TRAVERSAL " << endl;

inorder(root);

system("pause");

return;

}



* **POSTORDER TRAVERSING:**

Postorder traversal in Binary Search Tree (BST) is a traversal technique that visits the left subtree, then the right subtree, and finally the root node. This traversal is used to delete the nodes of the tree in the correct order.

#include"stdafx.h"

#include<iostream>

using namespace std;

class BST

{

public:

int data;

BST\*leftchild;

BST\*rightchild;

};

BST\*getvalue(BST\*root\_ref, int value)

{

BST\*newnode = NULL;

newnode = new BST();

newnode->data = value;

newnode->leftchild = NULL;

newnode->rightchild = NULL;

return newnode;

}

BST\*insertion(BST\*root,int value)

{

if (root == NULL)

{

return getvalue(root,value);

}

if (root->data > value)

{

root->leftchild = insertion(root->leftchild, value);

}

else if (root->data < value)

{

root->rightchild = insertion(root->rightchild, value);

}

return root;

}

void postorder(BST \*root)

{

if (root == NULL)

{

return;

}

postorder(root->leftchild);

postorder(root->rightchild);

cout << root->data << endl;

}

void main()

{

BST\*root = NULL;

int value;

root=getvalue(root, 2);

insertion(root, 1);

insertion(root, 4);

insertion(root, 8);

insertion(root, 3);

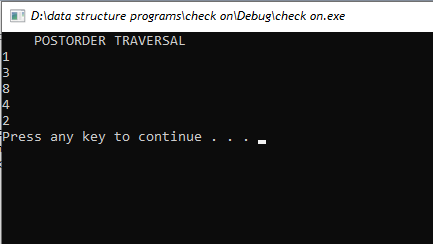
cout << " POSTORDER TRAVERSAL " << endl;

postorder(root);

system("pause");

return;

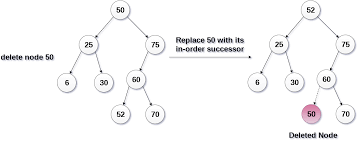
}



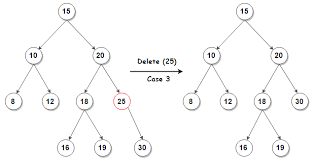
1. **DELETION:**

Deletion in a binary search tree is a process of removing a node from the tree while maintaining the structural integrity of the tree. It is a recursive process and the algorithm is similar to that of the binary search algorithm. It involves locating the node to be deleted and replacing it with a suitable node from its subtree.

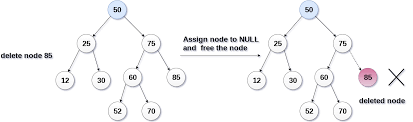
**Deleting leaf node:**



**Deleting node having single child:**



**Deleting root node:**



#include"stdafx.h"

#include<iostream>

using namespace std;

class BST

{

public:

int data;

BST\*leftchild;

BST\*rightchild;

};

BST\*getvalue(BST\*root\_ref, int value)

{

BST\*newnode = NULL;

newnode = new BST();

newnode->data = value;

newnode->leftchild = NULL;

newnode->rightchild = NULL;

return newnode;

}

BST\*insertion(BST\*root,int value)

{

if (root == NULL)

{

return getvalue(root,value);

}

if (root->data > value)

{

root->leftchild = insertion(root->leftchild, value);

}

else if (root->data < value)

{

root->rightchild = insertion(root->rightchild, value);

}

return root;

}

void inorder(BST \*root)

{

if (root == NULL)

{

return;

}

inorder(root->leftchild);

cout << root->data << endl;

inorder(root->rightchild);

}

BST\* delete\_min(BST\*root)

{

BST \*current = root;

while (current->leftchild != NULL)

{

current = current->leftchild;

}

return current;

}

BST\* deletee(BST\* root,int key)

{

if (root == NULL)

{

return root;

}

if (key > root->data)

{

root->rightchild = deletee( root-> rightchild,key);

}

else if (key < root->data)

{

root->leftchild = deletee( root-> leftchild,key);

}

else{

if (root->leftchild == NULL)

{

BST \*temp = root->rightchild;

delete(root);

return temp;

}

else if (root->rightchild == NULL)

{

BST\*temp = root->leftchild;

delete(root);

return(temp);

}

else{

BST\*temp = delete\_min(root->rightchild);

root->data = temp->data;

root->rightchild = deletee(root->rightchild, temp->data);

}

}

return root;

}

void main()

{

BST\*root = NULL;

int value;

root=getvalue(root, 2);

insertion(root, 1);

insertion(root, 4);

insertion(root, 8);

insertion(root, 3);

cout << " LIST BEOFRE DELETION " << endl;

inorder(root); //1,2,3,4,8

int key;

cout << " Enter data to delete " << endl;

cin >> key;

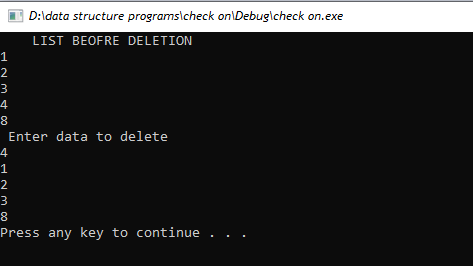
root=deletee(root,key);

inorder(root);

system("pause");

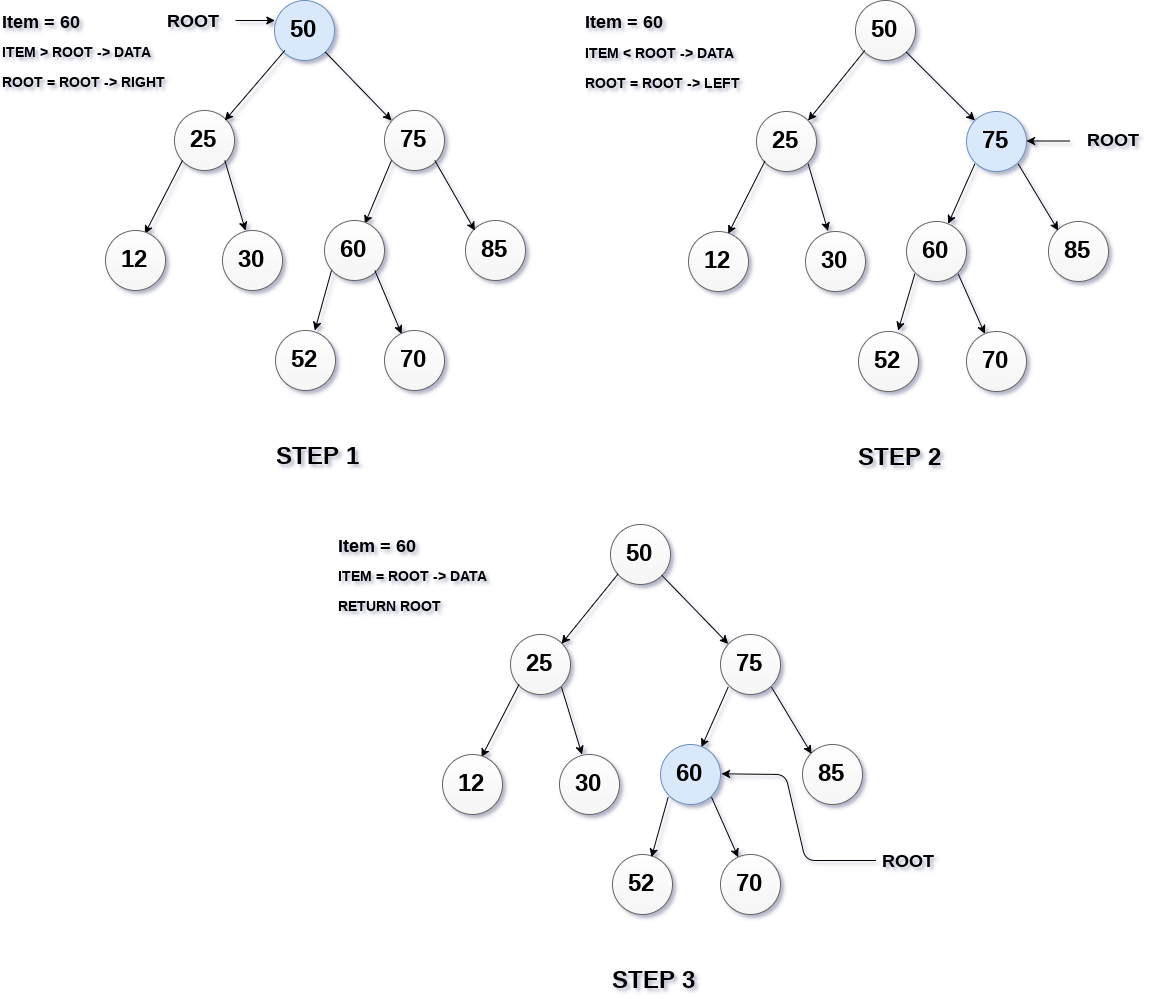
return;

}



1. **SEARCHING:**

Binary Search Tree (BST) is a special type of data structure used to store data in a tree-like structure. The data is organized in such a way that searches become easier and faster. It works by storing data in nodes, which are connected to each other in a tree structure. The data is sorted in an ascending order, which makes it easier to search for a specific item. The search begins at the root node and then proceeds to the left or right side of the tree based on the value being searched for. This makes finding the desired item faster and more efficient.



#include"stdafx.h"

#include<iostream>

using namespace std;

class BST

{

public:

int data;

BST\*leftchild;

BST\*rightchild;

};

BST\*getvalue(BST\*root\_ref, int value)

{

BST\*newnode = NULL;

newnode = new BST();

newnode->data = value;

newnode->leftchild = NULL;

newnode->rightchild = NULL;

return newnode;

}

BST\*insertion(BST\*root,int value)

{

if (root == NULL)

{

return getvalue(root,value);

}

if (root->data > value)

{

root->leftchild = insertion(root->leftchild, value);

}

else if (root->data < value)

{

root->rightchild = insertion(root->rightchild, value);

}

return root;

}

void preorder(BST \*root)

{

if (root == NULL)

{

return;

}

cout << root->data << endl;

preorder(root->leftchild);

preorder(root->rightchild);

}

void postorder(BST \*root)

{

if (root == NULL)

{

return;

}

postorder(root->leftchild);

postorder(root->rightchild);

cout << root->data << endl;

}

void inorder(BST \*root)

{

if (root == NULL)

{

return;

}

inorder(root->leftchild);

cout << root->data << endl;

inorder(root->rightchild);

}

BST\* search(BST\*root, int value)

{

if (root == NULL || root->data == value)

{

return root;

}

if (value > root->data)

{

return search(root->rightchild, value);

}

if (value < root->data)

{

return search(root->leftchild, value);

}

}

void main()

{

BST\*root = NULL;

int value;

root=getvalue(root, 2);

insertion(root, 1);

insertion(root, 4);

insertion(root, 8);

insertion(root, 3);

cout << " INORDER TRAVERSAL " << endl;

inorder(root);

cout << " Enter the value u want to search from the tree " << endl;

cin >> value;

cout << " SEARCHING IN TREE " << endl;

BST\* temp=search(root,value);

if (temp == NULL)

{

cout << " Value not found........! " << endl;

}

else

{

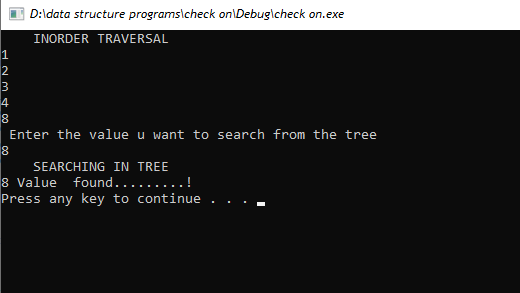
cout <<temp->data << " Value found.........! " << endl;

}

system("pause");

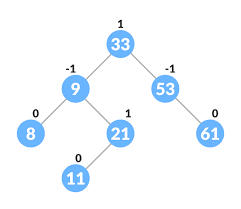
return;

}



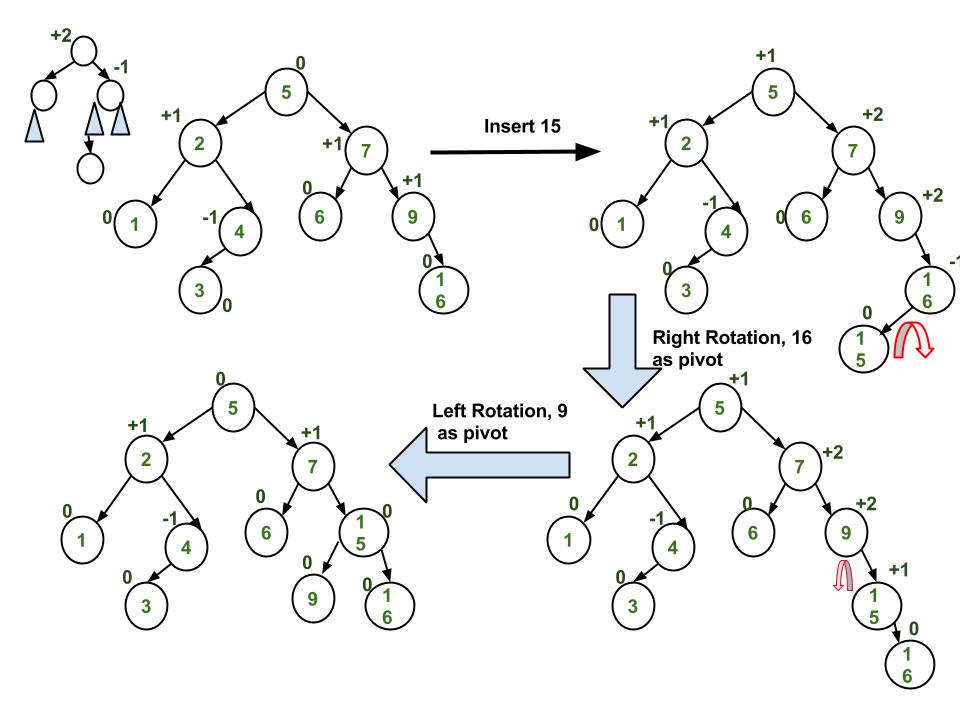
1. **AVL TRESS:**

An AVL tree is a self-balancing binary search tree. In an AVL tree, the heights of the two child subtrees of any node differ by at most one; if at any time they differ by more than one, rebalancing is done to restore this property.



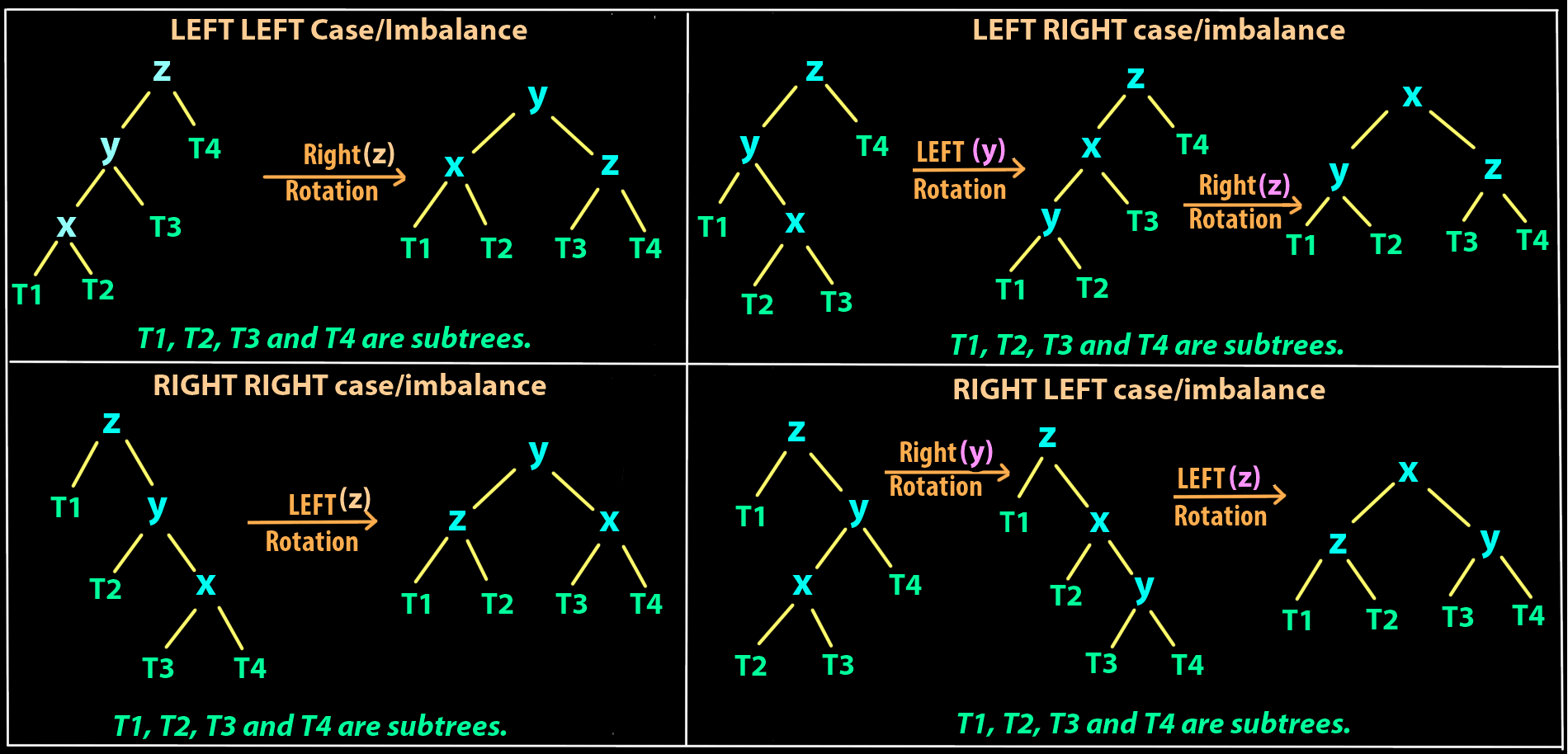
1. **INSERTION:**

An AVL tree is a self-balancing binary search tree. Insertion into an AVL tree is a process of inserting a new node into the tree while maintaining the balance of the tree. It is done by performing necessary rotations after insertion to ensure that the tree remains balanced.



**Rotations:**

Rotation in an AVL tree is an operation used to maintain the balance of a tree. It is applied to a node, and consists of rearranging the links between the node and its children and/or its parent to preserve the AVL tree properties of the height difference between two sub-trees of any node being at most 1. Rotation is used to ensure that the tree remains balanced after an insertion or deletion.



#include "stdafx.h"

#include <iostream>

#include<string>

using namespace std;

class Node {

public:

int key;

Node \*left;

Node \*right;

int height;

};

int max(int a, int b);

// Calculate height

int height(Node \*N) {

if (N == NULL)

return 0;

return N -> height;

}

int max(int a, int b) {

return (a > b) ? a : b;

}

// New node creation

Node \*newNode(int key) {

Node \*node = new Node();

node -> key = key;

node -> left = NULL;

node -> right = NULL;

node -> height = 1;

return (node);

}

// Rotate right

Node \*rightRotate(Node \*y) {

Node \*x = y -> left;

Node \*T2 = x -> right;

x -> right = y;

y -> left = T2;

y -> height = max(height(y -> left),height(y -> right)) +1;

x -> height = max(height(x -> left),height(x -> right)) +1;

return x;

}

// Rotate left

Node \*leftRotate(Node \*x) {

Node \*y = x -> right;

Node \*T2 = y -> left;

y -> left = x;

x -> right = T2;

x -> height = max(height(x -> left),height(x -> right)) +1;

y -> height = max(height(y -> left),height(y -> right)) + 1;

return y;

}

// Get the balance factor of each node

int getBalanceFactor(Node \*N) {

if (N == NULL)

return 0;

return height(N -> left) - height(N -> right);

}

// Insert a node

Node \*insertNode(Node \*node, int key) {

// Find the correct postion and insert the node

if (node == NULL)

return (newNode(key));

if (key < node -> key)

node -> left = insertNode(node -> left, key);

else if (key > node -> key)

node -> right = insertNode(node -> right, key);

else

return node;

// Update the balance factor of each node and

// balance the tree

node -> height = 1 + max(height(node -> left), height(node -> right));

int balanceFactor = getBalanceFactor(node);

if (balanceFactor > 1 && key > node -> left -> key)

{

return rightRotate(node);

}

if (balanceFactor > 1 && key > node -> left -> key)

{

node -> left = leftRotate(node -> left);

return rightRotate(node);

}

if (balanceFactor < -1 && key > node -> right -> key)

{

return leftRotate(node);

}

if (balanceFactor < -1 && key < node -> right -> key) {

node -> right = rightRotate(node -> right);

return leftRotate(node);

}

return node;

}

// Node with minimum value

Node \*nodeWithMimumValue(Node \*node)

{

Node \*current = node;

while (current ->left != NULL)

current = current -> left;

return current;

}

// Print the tree

void printTree(Node \*root, string indent,bool last)

{

if (root != nullptr) {

cout << indent;

if (last) {

cout <<" R---- ";

indent += " ";

}

else {

cout <<" L----";

indent += " | ";

}

cout<< root -> key << endl;

printTree(root -> left, indent, false);

printTree(root -> right, indent, true);

}

}

int main() {

Node \*root = NULL;

root = insertNode(root, 33);

root = insertNode(root, 13);

root = insertNode(root, 53);

root = insertNode(root, 9);

root = insertNode(root, 21);

root = insertNode(root, 61);

root = insertNode(root, 8);

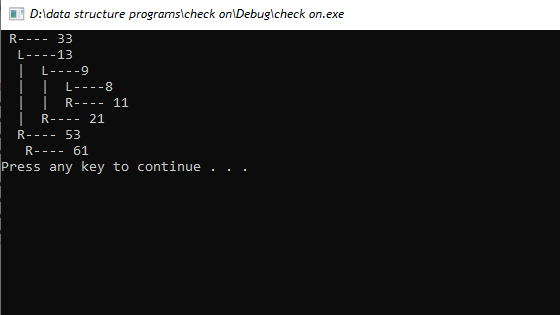
root = insertNode(root, 11);

printTree(root, "" , true);

system("pause");

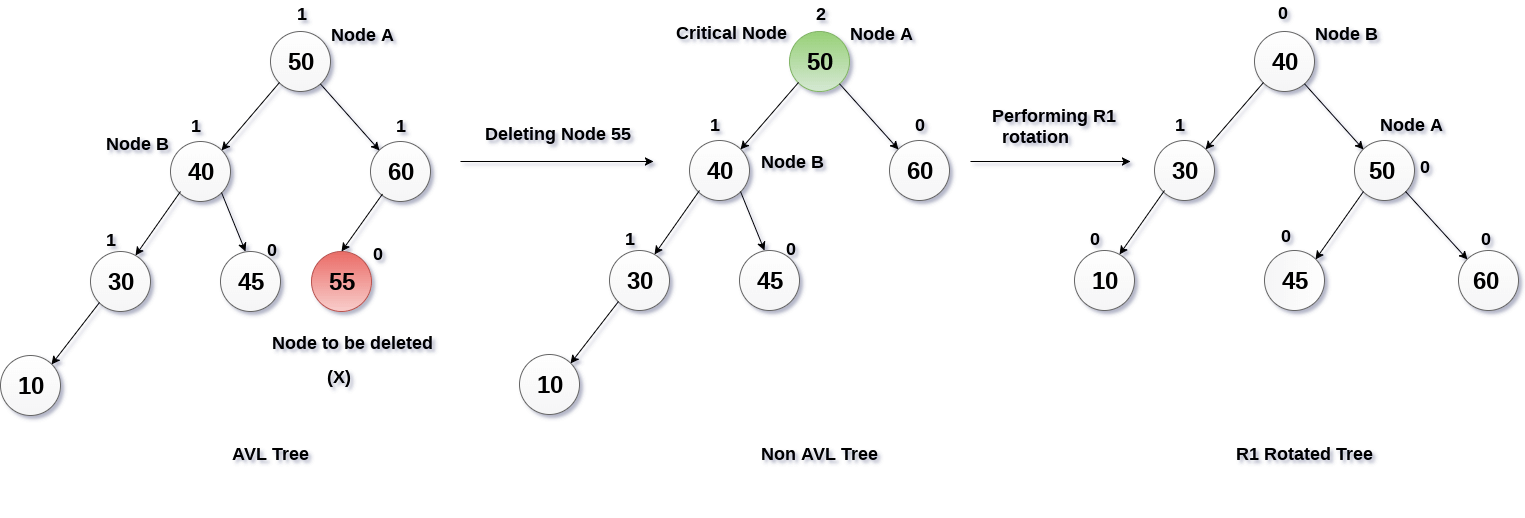
return 0;

}



1. **DELETION:**

Deletion in an AVL tree is the process of removing a node from the tree while maintaining its balance. This is done by either replacing the node with its in-order predecessor or successor, or by performing a series of rotations in order to maintain the tree balance.



#include "stdafx.h"

#include <iostream>

#include<string>

using namespace std;

class Node {

public:

int key;

Node \*left;

Node \*right;

int height;

};

int max(int a, int b);

// Calculate height

int height(Node \*N) {

if (N == NULL)

return 0;

return N -> height;

}

int max(int a, int b) {

return (a > b) ? a : b;

}

// New node creation

Node \*newNode(int key) {

Node \*node = new Node();

node -> key = key;

node -> left = NULL;

node -> right = NULL;

node -> height = 1;

return (node);

}

// Rotate right

Node \*rightRotate(Node \*y) {

Node \*x = y -> left;

Node \*T2 = x -> right;

x -> right = y;

y -> left = T2;

y -> height = max(height(y -> left),height(y -> right)) +1;

x -> height = max(height(x -> left),height(x -> right)) +1;

return x;

}

// Rotate left

Node \*leftRotate(Node \*x) {

Node \*y = x -> right;

Node \*T2 = y -> left;

y -> left = x;

x -> right = T2;

x -> height = max(height(x -> left),height(x -> right)) +1;

y -> height = max(height(y -> left),height(y -> right)) + 1;

return y;

}

// Get the balance factor of each node

int getBalanceFactor(Node \*N) {

if (N == NULL)

return 0;

return height(N -> left) - height(N -> right);

}

// Insert a node

Node \*insertNode(Node \*node, int key) {

// Find the correct postion and insert the node

if (node == NULL)

return (newNode(key));

if (key < node -> key)

node -> left = insertNode(node -> left, key);

else if (key > node -> key)

node -> right = insertNode(node -> right, key);

else

return node;

// Update the balance factor of each node and

// balance the tree

node -> height = 1 + max(height(node -> left), height(node -> right));

int balanceFactor = getBalanceFactor(node);

if (balanceFactor > 1 && key > node -> left -> key)

{

return rightRotate(node);

}

if (balanceFactor > 1 && key > node -> left -> key)

{

node -> left = leftRotate(node -> left);

return rightRotate(node);

}

if (balanceFactor < -1 && key > node -> right -> key)

{

return leftRotate(node);

}

if (balanceFactor < -1 && key < node -> right -> key) {

node -> right = rightRotate(node -> right);

return leftRotate(node);

}

return node;

}

// Node with minimum value

Node \*nodeWithMimumValue(Node \*node)

{

Node \*current = node;

while (current ->left != NULL)

current = current -> left;

return current;

}

// Delete a node

Node \*deleteNode(Node \*root, int key) {

// Find the node and delete it

if (root == NULL)

return root;

if (key < root -> key)

root -> left = deleteNode(root -> left, key);

else if (key > root -> key)

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

else {

if ((root -> left == NULL) ||(root -> right == NULL))

{

Node \*temp = root -> left ? root -> left : root -> right;

if (temp == NULL) {

temp = root;

root = NULL;

}

else

\*root = \*temp;

free(temp);

}

else {

Node \*temp =

nodeWithMimumValue(root -> right);

root -> key = temp -> key;

root -> right = deleteNode(root -> right,temp -> key);

}

}

if (root == NULL)

return root;

// Update the balance factor of each node

// balance the tree

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

int balanceFactor = getBalanceFactor(root);

if (balanceFactor > 1) {

if (getBalanceFactor(root -> left) >= 0) {

return rightRotate(root);

}

else {

root -> left = leftRotate(root -> left);

return rightRotate(root);

}

}

if (balanceFactor < -1) {

if (getBalanceFactor(root -> right) <= 0) {

return leftRotate(root);

}

else {

root -> right = rightRotate(root -> right);

return leftRotate(root);

}

}

return root;

}

// Print the tree

void printTree(Node \*root, string indent,bool last)

{

if (root != nullptr) {

cout << indent;

if (last) {

cout <<" R---- ";

indent += " ";

}

else {

cout <<" L----";

indent += " | ";

}

cout<< root -> key << endl;

printTree(root -> left, indent, false);

printTree(root -> right, indent, true);

}

}

int main() {

Node \*root = NULL;

root = insertNode(root, 33);

root = insertNode(root, 13);

root = insertNode(root, 53);

root = insertNode(root, 9);

root = insertNode(root, 21);

root = insertNode(root, 61);

root = insertNode(root, 8);

root = insertNode(root, 11);

printTree(root, "" , true);

root = deleteNode(root, 13);

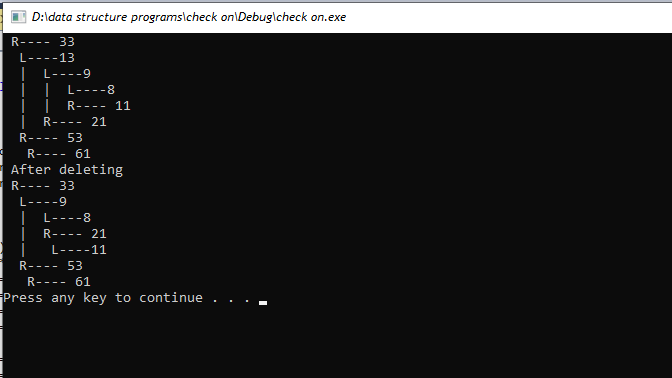
cout << " After deleting "<< endl;

printTree(root, "" , true);

system("pause");

return 0;

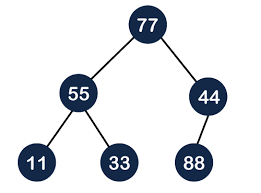
}



1. **SEARCHING:**

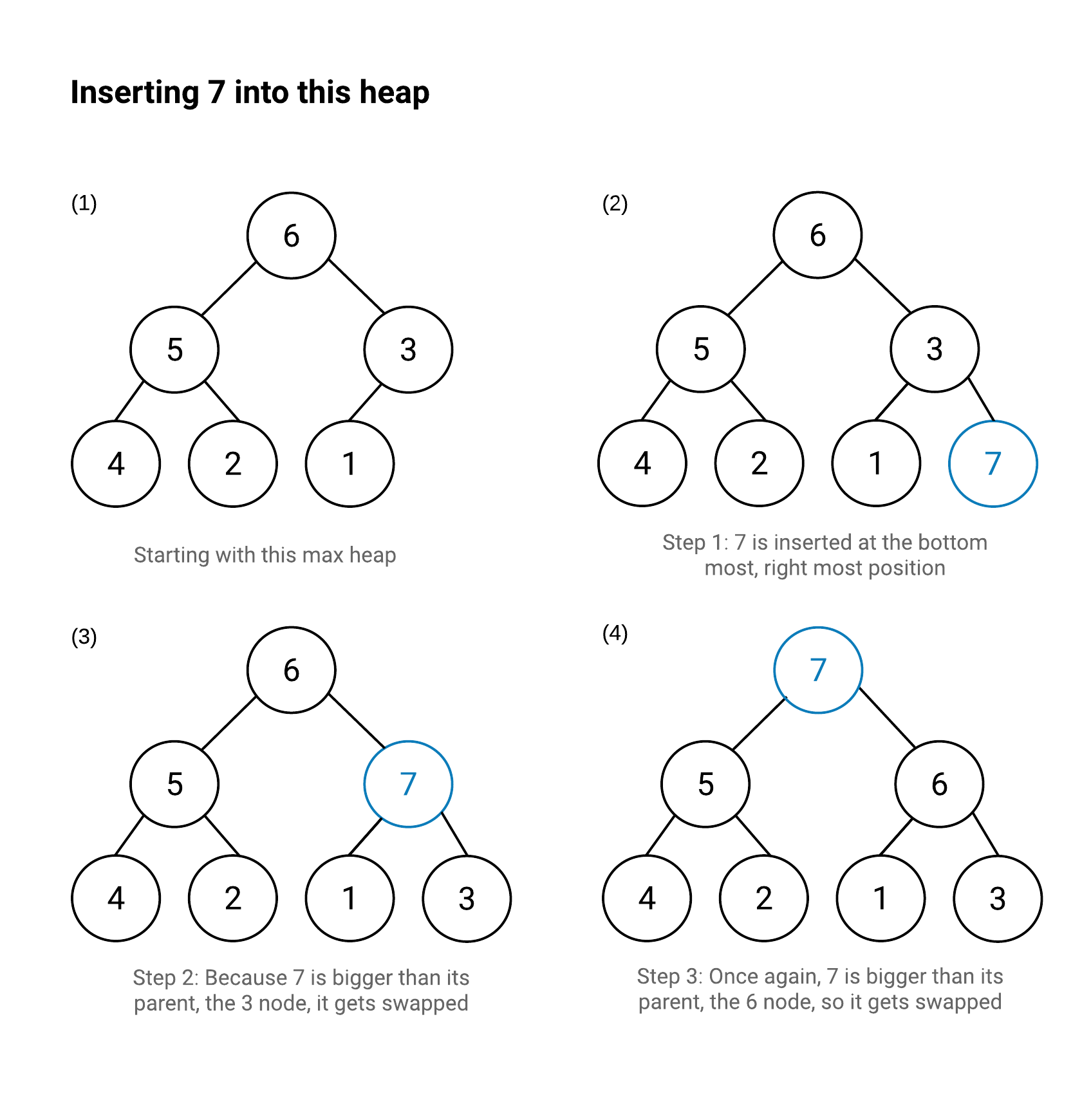
AVL tree is a self-balancing binary search tree that maintains a balance between the heights of its left and right subtrees, allowing for efficient search, insertion, and deletion operations. It uses a rotational mechanism to keep the tree balanced, which increases the overall time complexity of the operations.

1. **HEAPS:**
2. **MAX HEAP:**  
   A Max Heap is a type of binary heap data structure that satisfies the heap property: the value of any node is greater than or equal to the value of its parent node. This means that the root node of a Max Heap will always have the largest value in the heap.



1. **INSERTION:**

Insertion in a max heap is the process of adding a new element to the heap while maintaining the max heap property. The new element is inserted at the bottom of the heap and then the max heap property is restored by comparing the inserted element with its parent and swapping them if the parent is smaller. This process is repeated until the max heap property is satisfied.



Code:

#include "stdafx.h"

#include <iostream>

#include <vector>

using namespace std;

void swap(int \*a, int \*b)

{

int temp = \*b;

\*b = \*a;

\*a = temp;

}

void heapify(vector<int> &hT, int i)

{

int size = hT.size();

int largest = i;

int l = 2 \* i + 1;

int r = 2 \* i + 2;

if (l < size && hT[l] > hT[largest])

largest = l;

if (r < size && hT[r] > hT[largest])

largest = r;

if (largest != i)

{

swap(&hT[i], &hT[largest]);

heapify(hT, largest);

}

}

void insert(vector<int> &hT, int newNum)

{

int size = hT.size();

if (size == 0)

{

hT.push\_back(newNum);

}

else

{

hT.push\_back(newNum);

for (int i = size / 2 - 1; i >= 0; i--)

{

heapify(hT, i);

}

}

}

void printArray(vector<int> &hT)

{

for (int i = 0; i < hT.size(); i++)

{

cout << hT[i] << endl;

}

cout << "\n";

}

int main()

{

vector<int> heapTree;

insert(heapTree, 3);

insert(heapTree, 4);

insert(heapTree, 9);

insert(heapTree, 5);

insert(heapTree, 2);

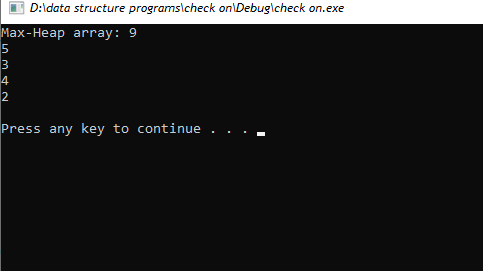
cout << "Max-Heap array: ";

printArray(heapTree);

system("pause");

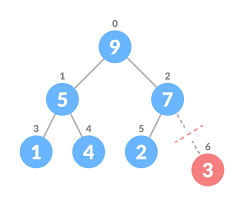
return 0;

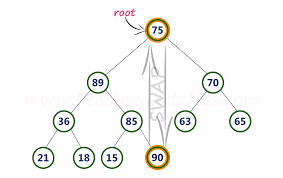
}



1. **DELETION:**

Deletion in a Max Heap is the process of removing a node from the heap. This is done by first replacing the node to be deleted with the last node in the heap, and then using heapify to maintain the properties of the heap. The deleted node can be returned or discarded afterwards.





#include "stdafx.h"

#include <iostream>

#include <vector>

using namespace std;

void swap(int \*a, int \*b)

{

int temp = \*b;

\*b = \*a;

\*a = temp;

}

void heapify(vector<int> &hT, int i)

{

int size = hT.size();

int largest = i;

int l = 2 \* i + 1;

int r = 2 \* i + 2;

if (l < size && hT[l] > hT[largest])

largest = l;

if (r < size && hT[r] > hT[largest])

largest = r;

if (largest != i)

{

swap(&hT[i], &hT[largest]);

heapify(hT, largest);

}

}

void insert(vector<int> &hT, int newNum)

{

int size = hT.size();

if (size == 0)

{

hT.push\_back(newNum);

}

else

{

hT.push\_back(newNum);

for (int i = size / 2 - 1; i >= 0; i--)

{

heapify(hT, i);

}

}

}

void deleteNode(vector<int> &hT, int num)

{

int size = hT.size();

int i;

for (i = 0; i < size; i++)

{

if (num == hT[i])

break;

}

swap(&hT[i], &hT[size - 1]);

hT.pop\_back();

for (int i = size / 2 - 1; i >= 0; i--)

{

heapify(hT, i);

}

}

void printArray(vector<int> &hT)

{

for (int i = 0; i < hT.size(); i++)

{

cout << hT[i] << endl;

}

cout << "\n";

}

int main()

{

vector<int> heapTree;

insert(heapTree, 3);

insert(heapTree, 4);

insert(heapTree, 9);

insert(heapTree, 5);

insert(heapTree, 2);

cout << "Max-Heap array: ";

printArray(heapTree);

deleteNode(heapTree, 4);

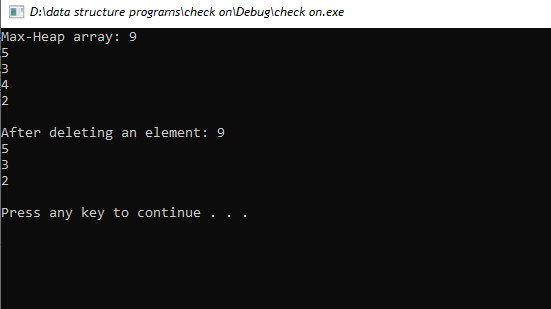
cout << "After deleting an element: ";

printArray(heapTree);

system("pause");

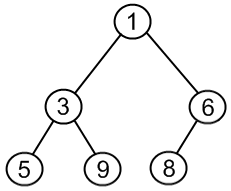
return 0;

}



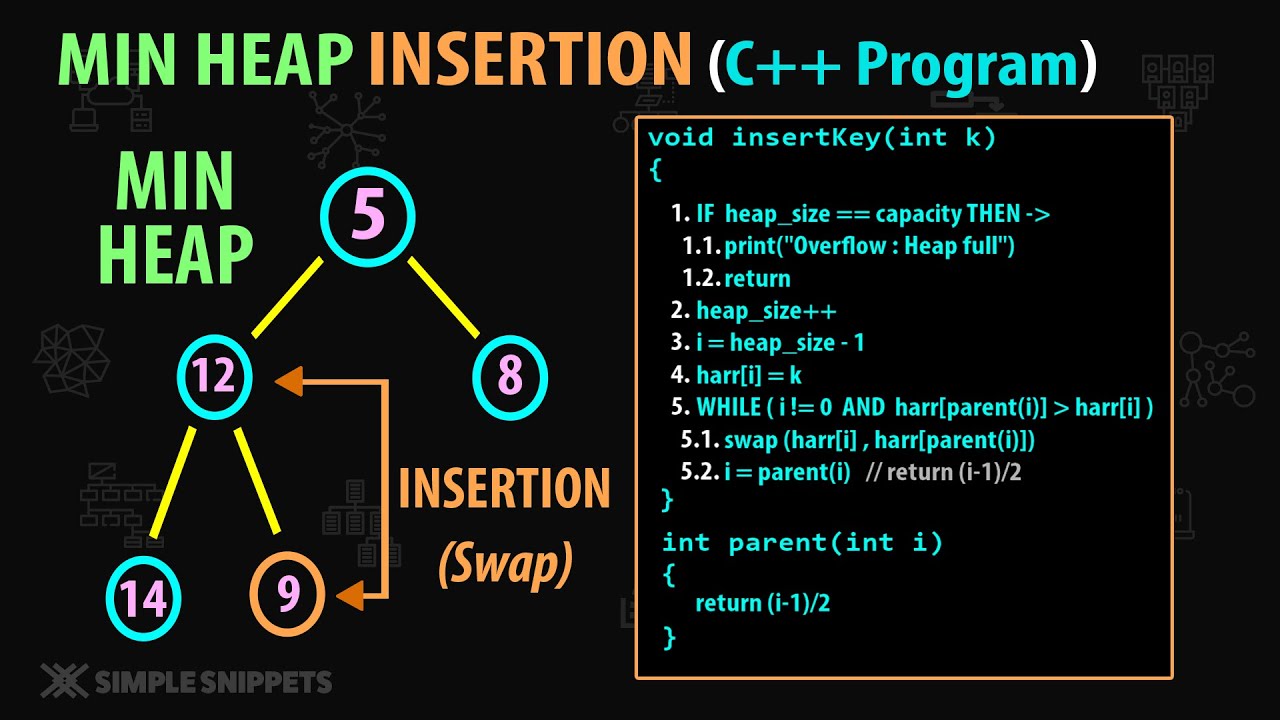
1. **MIN HEAP :**

Min Heap is a type of binary heap data structure which has the property that the root node is the minimum of all the nodes in the heap. It is typically used for priority queues and sorting algorithms.



1. **INSERTION:**

Insertion in min heap is the process of adding a new element to a min heap, maintaining the min heap order property. It involves first inserting the new element at the end of the heap and then comparing it with its parent until it finds the correct position in the heap.



#include "stdafx.h"

#include <iostream>

#include <vector>

using namespace std;

void swap(int \*a, int \*b)

{

int temp = \*b;

\*b = \*a;

\*a = temp;

}

void heapify(vector<int> &hT, int i)

{

int size = hT.size();

int largest = i;

int l = 2 \* i + 1;

int r = 2 \* i + 2;

if (l < size && hT[l] < hT[largest])

largest = l;

if (r < size && hT[r] < hT[largest])

largest = r;

if (largest != i)

{

swap(&hT[i], &hT[largest]);

heapify(hT, largest);

}

}

void insert(vector<int> &hT, int newNum)

{

int size = hT.size();

if (size == 0)

{

hT.push\_back(newNum);

}

else

{

hT.push\_back(newNum);

for (int i = size / 2 - 1; i >= 0; i--)

{

heapify(hT, i);

}

}

}

void printArray(vector<int> &hT)

{

for (int i = 0; i < hT.size(); i++)

{

cout << hT[i] << endl;

}

cout << "\n";

}

int main()

{

vector<int> heapTree;

insert(heapTree, 3);

insert(heapTree, 4);

insert(heapTree, 9);

insert(heapTree, 5);

insert(heapTree, 2);

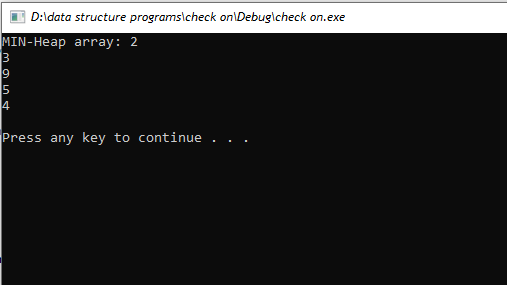
cout << "MIN-Heap array: ";

printArray(heapTree);

system("pause");

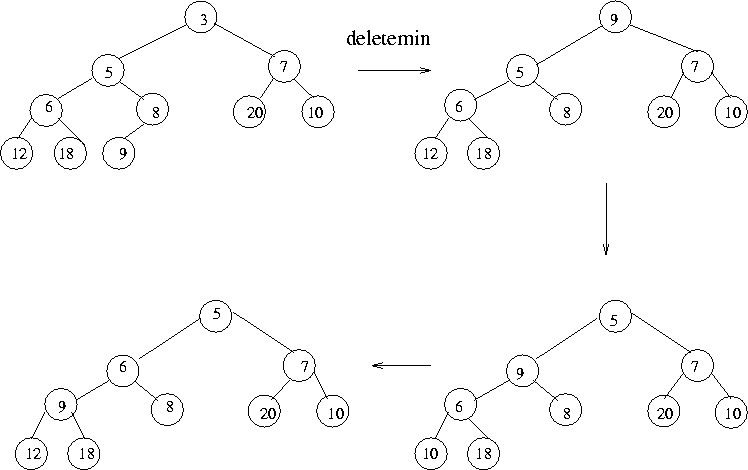
return 0;

}



1. **DELETION:**

Deletion in min heap is the process of removing the root node from the heap and replacing it with the last element in the heap. This ensures that the heap remains a valid min heap after the deletion.



#include "stdafx.h"

#include <iostream>

#include <vector>

using namespace std;

void swap(int \*a, int \*b)

{

int temp = \*b;

\*b = \*a;

\*a = temp;

}

void heapify(vector<int> &hT, int i)

{

int size = hT.size();

int largest = i;

int l = 2 \* i + 1;

int r = 2 \* i + 2;

if (l < size && hT[l] < hT[largest])

largest = l;

if (r < size && hT[r] < hT[largest])

largest = r;

if (largest != i)

{

swap(&hT[i], &hT[largest]);

heapify(hT, largest);

}

}

void insert(vector<int> &hT, int newNum)

{

int size = hT.size();

if (size == 0)

{

hT.push\_back(newNum);

}

else

{

hT.push\_back(newNum);

for (int i = size / 2 - 1; i >= 0; i--)

{

heapify(hT, i);

}

}

}

void deleteNode(vector<int> &hT, int num)

{

int size = hT.size();

int i;

for (i = 0; i < size; i++)

{

if (num == hT[i])

break;

}

swap(&hT[i], &hT[size - 1]);

hT.pop\_back();

for (int i = size / 2 - 1; i >= 0; i--)

{

heapify(hT, i);

}

}

void printArray(vector<int> &hT)

{

for (int i = 0; i < hT.size(); i++)

{

cout << hT[i] << endl;

}

cout << "\n";

}

int main()

{

vector<int> heapTree;

insert(heapTree, 3);

insert(heapTree, 4);

insert(heapTree, 9);

insert(heapTree, 5);

insert(heapTree, 2);

cout << "MIN-Heap array: ";

printArray(heapTree);

deleteNode(heapTree, 4);

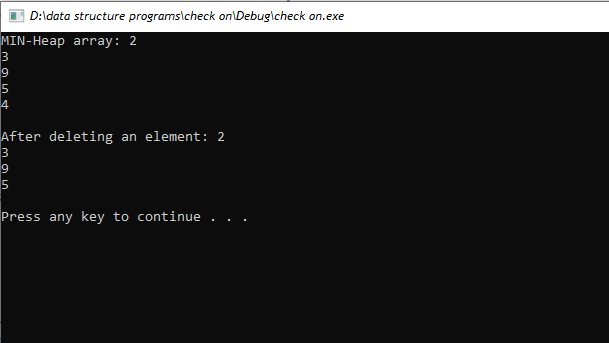
cout << "After deleting an element: ";

printArray(heapTree);

system("pause");

return 0;

}



**QUICK SORTING:**

Quick sort is also known as **Partition-exchange sort** based on the rule of **Divide and Conquer.**

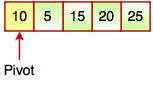
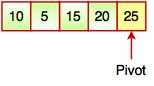
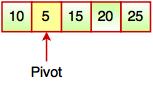
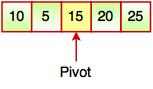
It is a highly efficient sorting algorithm.

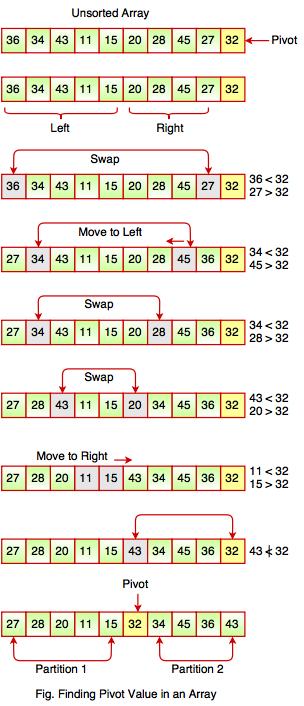
Quick sort is the quickest comparison-based sorting algorithm.

It is very fast and requires less additional space, only O(n log n) space is required.

Quick sort picks an element as pivot and partitions the array around the picked pivot.

**There are different versions of quick sort which choose the pivot in different ways:**

**1. First element as pivot**  
  
   
  
**2. Last element as pivot**  
  
   
  
**3. Random element as pivot**  
  
   
  
**4. Median as pivot**  
  
 



#include "stdafx.h"

#include <iostream>

#include<string>

using namespace std;

void swap(string \*a, string \*b) {

string temp = \*a;

\*a = \*b;

\*b = temp;

}

void printArray(string array[], int size) {

int i;

for (i = 0; i < size; i++)

cout<<"\t" << array[i] << " ";

cout << endl;

}

int partition(string array[], int low, int high) {

string pivot = array[high];

int p = (low - 1);

for (int q = low; q < high; q++) {

if (array[q] <= pivot) {

p++;

swap(&array[p], &array[q]);

}

}

swap(&array[p + 1], &array[high]);

return (p + 1);

}

void quickSort(string array[], int low, int high) {

if (low < high) {

int pi = partition(array, low, high);

quickSort(array, low, pi - 1);

quickSort(array, pi + 1, high);

}

}

int main() {

string data[] = { "Areeba","Rida","Ayesha","Laiba","Irum"};

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

cout << " Unsorted Array: \n"<<endl;

printArray(data, n);

quickSort(data, 0, n - 1);

cout << "\n Sorted array in ascending order: \n"<<endl;

printArray(data, n);

system("pause");

return 0;

}

