1. Using circular representation for a polynomial, design, develop, and execute a program in C to accept two polynomials, add them, and then print the resulting polynomial.

**STRUCTURE OF EACH NODE**

```c
struct polynomial{
    float coeff;
    int exponent;
    struct polynomial *link;
};
```

**typedef struct polynomial *NODEPTR;**

**GETNODE FUNCTION**

```c
NODE getnode(void)
```

Using the builtin malloc( ) function return a pointer to a chunk of memory whose size is same as that of the node.

**Adding two Polynomials**

Create a circular list with three header nodes. The idea is to store the terms of first polynomial using nodes stored between the first and second header nodes, the terms of second polynomial in between the second and third header nodes, and finally the terms in resultant polynomial in between the third and first header nodes, thereby making the list circular.

Assumption: The terms in each polynomial are read in the descending order of their exponents.

1. Read and store the terms in the first polynomial.
2. Read and store the terms in the second polynomial.
3. Assign pointers tracker1 and tracker2 to the first nodes of each polynomial respectively.
4. Repeat the following operations till either of the pointers reach the end of the polynomial
   1. if value pointed to by tracker1 has a greater exponent than that pointed by tracker2
      1. copy this term pointed by tracker1 to the resultant polynomial.
      2. advance tracker1 to the next node.
2. else if value pointed to by tracker2 has a greater exponent than that pointed by tracker1
   1. copy this term pointed by tracker2 to the resultant polynomial.
   2. advance tracker2 to the next node.
3. else
   1. copy the sum of the terms pointed by tracker1 and tracker2 to the resultant polynomial.
   2. advance tracker1 to the next node.
   3. advance tracker2 to the next node.
5. Display terms of the first polynomial.
6. Display terms of the second polynomial.
7. Display terms of the resultant polynomial.

2. Design, develop, and execute a program in C to convert a given valid parenthesized infix arithmetic expression to postfix expression and then to print both the expressions. The expression consists of single character operands and the binary operators + (plus), - (minus), * (multiply) and / (divide).

CONVERSION OF INFIX EXPRESSION TO POSTFIX EXPRESSION

1. Read the infix expression as a string.
2. Scan the expression character by character till the end. Repeat the following operations
   a. If it is an operand add it to the postfix expression.
   b. If it is a left parentheses push it onto the stack.
   3. If it is a right parentheses pop out elements from the stack and assign it to the postfix string, pop out the left parentheses but don’t assign to postfix.
   4. If it is a operator compare its precedence with that of the element at the top of stack.
      1. If it is greater push it onto the stack.
      2. Else pop and assign elements in the stack to the postfix expression until you find one such element.
8. If you have reached the end of the expression, pop out any leftover elements in the stack till it becomes empty.
9. Append a null terminator at the end display the result.

3. Design, develop, and execute a program in C to evaluate a valid postfix expression using stack. Assume that the postfix expression is read as a single line consisting of non-negative single digit operands and binary arithmetic operators. The arithmetic operators are +(add), - (subtract), * (multiply) and / (divide).

EVALUATION OF POSTFIX EXPRESSION

1. Read the infix expression as a string.
2. Scan the expression character by character till the end. Repeat the following operations
   a. If it is an operand push it onto the stack.
   b. If it is an operator
1. Pop out two operands from stack
2. Apply the operator onto the popped operands.
3. Store the result back on to the stack.
3. On reaching the end of expression pop out the contents of the stack and display as the result.

4. Design, develop, and execute a program in C to simulate the working of a queue of integers using an array. Provide the following operations:
   a. Insert
   b. Delete
   c. Display

GLOBAL SECTION
1. Define size of queue.
2. Allocate required array memory.
3. Initialise pointers front to 0 and rear to -1.

IMPLEMENTING THE INSERT_REAR FUNCTION
1. Check whether queue is full.
2. If yes display an error message.
3. Else increment rear pointer and place the element to be pushed at that position.

IMPLEMENTING THE DELETE_FRONT FUNCTION
1. Check whether queue is empty.
2. If yes display an error message.
3. Else delete the element pointed to by the front pointer then increment front pointer.

IMPLEMENTING THE DISPLAY FUNCTION
1. Check whether queue is empty.
2. If yes display an error message.
3. Else display elements from front to rear.

5. Design, develop, and execute a program in C++ based on the following requirements: An EMPLOYEE class is to contain the following data members and member functions: Data members: Employee_Number (an integer), Employee_Name (a string of characters), Basic_Salary (an integer), All_Allowances (an integer), IT (an integer), Net_Salary (an integer). Member functions: to read the data of an employee, to calculate Net_Salary and to print the values of all the data members. (All_Allowances = 123% of Basic; Income Tax (IT) = 30% of the gross salary (= basic_Salary + All_Allowance); Net_Salary = Basic_Salary + All_Allowances – IT)
#include<iostream.h>
#include<conio.h>

class employee
{
    int num;
    char name[20];
    float basic,da,it,netsal;

public:
    void readdata();
    void netsalary();
    void display();
    int getnum();
};

void employee::readdata()
{
    cout<<"\n Enter employee number: ";
    cin>>num;
    cout<<"\n Enter name: ";
    cin>>name;
    cout<<"\n Enter basic: ";
    cin>>basic;
}

void employee::netsalary()
{
    float sal;
    da=0.52*basic;
    sal=da+basic;
    it=0.3*sal;
    netsal=sal-it;
}
void employee::display()
{
    cout.precision(4);
    cout.setf(ios::right);
    cout<<"\n"<<num;
    cout.width(15);
    cout<<name;
    cout.width(10);
    cout<<basic;
    cout.width(10);
    cout<<da;
    cout.width(10);
    cout<<it;
    cout.width(10);
    cout<<netsal;
}

int employee::getnum()
{
    return num;
}

void main()
{
    int i,n;
    clrscr();
    cout<<"\n Enter the number of employees:";
    cin>>n;
employee e[20];
for(i=0;i<n;i++)
{
    cout<<"\n Enter the details of employee "+i+1;
    e[i].readdata();
    for(int j=0;j<i;j++)
    {
        if(e[i].getnum()==e[j].getnum())
        {
            cout<<"\n Duplicate entry of employee number, re-type";
            i--;
        }
    }
}
for(i=0;i<n;i++)
e[i].netsalary();
cout<<"\n NUMBER\t NAME\t BASIC\t DA\t IT\t NET SALARY\n";
for(i=0;i<n;i++)
e[i].display();
getch();

OUTPUT
Enter the number of employees: 3
Enter the details of employee 1
Enter employee number: 1
Enter name: aaa
Enter basic:100
Enter the details of employee 2
Enter employee number:2
Enter name:www
Enter basic:250
Enter the details of employee 3
Enter employee number:3
Enter name:rtyu
Enter basic:900

<table>
<thead>
<tr>
<th>NUMBER</th>
<th>NAME</th>
<th>BASIC</th>
<th>DA</th>
<th>IT</th>
<th>NET SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>aaa</td>
<td>100</td>
<td>52</td>
<td>45.6</td>
<td>106.4</td>
</tr>
<tr>
<td>2</td>
<td>www</td>
<td>250</td>
<td>130</td>
<td>114</td>
<td>266</td>
</tr>
<tr>
<td>3</td>
<td>rtyu</td>
<td>900</td>
<td>468</td>
<td>410.4</td>
<td>957.6</td>
</tr>
</tbody>
</table>

6. Design, develop, and execute a program in C++ to create a class called STRING and implement the following operations. Display the results after every operation by overloading the operator <<.
   i. STRING s1 = “VTU”
   ii. STRING s2 = “BELGAUM”
   iii. STRING s3 = s1 + s2; (Use copy constructor)

Why overload operators

Clarity. It's common to define a meaning for an existing operator for objects of a new class. Operators are defined as either member functions or friend functions. Don't use operator overloading just because it can be done and is a clever trick. The purpose of operator overloading is to make programs clearer by using conventional meanings for ==, [], +, etc.

For example, overloading the [] operator for a data structure allows x = v[25] in place of a function call. This is purely a conveniece to the user of a class. Operator overloading isn't strictly necessary unless other classes or functions expect operators to be defined (as is sometimes the case).

Overview

C++ allows almost all operators to be overloaded to apply to class objects. For example, the following code shows some operators (in red) being applied to objects of type Intlist:

   IntList L1, L2, L3;
L1 += 10;
L3 = L1 + L2;
cout << L1;
if (L1[0] == 5) ... 
if (L1 == L2) ...

It is important to note that if you do not define these operators to apply to your class, you will get a compile-time error if you write code that does apply them to class objects; the compiler supplies a default version only for operator= (assignment).

Note also that when you overload operators, at least one operand must be a class object (or an enum). For example, you can redefine the plus operator applied to an IntList and an integer, or to two IntLists, but you cannot redefine plus applied to two integers. You also cannot redefine the arity of an operator; for example, you cannot define plus to have three operands.

Some operators are unary (have one operand), and some are binary (have two operands). The example given above includes only binary operators: +=, +, <<, [], and ==. Examples of unary operators are: unary minus, pre- and post-increment, and pre- and post-decrement.

Unary operators, and binary operators whose left operands are class objects, can be defined either as member functions of that class, or as free functions. Binary operators whose left operands are not class objects must be defined as free functions.

Example: Overloading +=
In the statement:
L1 += 10;
the left operand of += is an IntList, and its right operand is an integer. Therefore, this version of += can be defined either as an IntList member function or as a free function. Here is how it might be defined as a member function:

class IntList {
  public:
    void operator+=( int n );
    ...
};

void IntList::operator+=( int n ) {
  AddToEnd( n );
}

Note that it is up to the designer of the class to decide what it means to apply an operator to a class object. This definition of operator+= adds the right-hand operand (the integer value) to the end of the list represented by the left-hand operand (the IntList whose member function is called). It would also be reasonable to define operator+= to add the integer to the front of the list.

Example: Overloading <<
In the statement:
   
   cout << L1;
the left operand of << is an ostream, not an *IntList*. Therefore, operator<< cannot be defined as a
member function of the *IntList* class; it must be defined as a free function. For example:

```cpp
ostream &operator<<( ostream &out, const IntList &L ) {
   L.Print(out);
   return out;
}
```

Note that operator<< should be defined to return an *ostream* (so that "chained" output, like: cout
<< L1 << endl will work). (It is returned by reference -- that's what the & means -- for
efficiency.) Also note that operator<<'s first parameter must be an *ostream* passed by reference.
Its second parameter, the *IntList* that is printed, does not have to be passed as a const-reference
parameter; however it is more efficient to pass it by reference than by value (since that avoids a
call to the copy constructor), and it should not be modified by operator<<, so it should be a const
reference parameter.

Note also that this code assumes that the *IntList* class has a *Print* function. If that is not true, then
operator<< must be declared to be a friend function of the *IntList* class, so that it has access to
the private *Items* and *numItems* fields (which it needs in order to be able to print the contents of
the list). In that case, the function would be declared like this:

```cpp
class IntList {
   friend ostream &operator<<( ostream &out, const IntList &L );

   public:
   ...
};
```

and defined like this:

```cpp
ostream &operator<<( ostream &out, const IntList &L ) {
   out << "[ ";
   for (int k=0; k< L.numItems; k++) {
      out << L.Items[k] << ' ';
   }
   out << "]";
}
```

Copy constructor

Copy constructor is a type of constructor which constructs an object by copying the state from
another object of the same class. Whenever an object is copied, another object is created and in
this process the copy constructor gets called. If the class of the object being copied is x, the copy
constructor’s signature is usually x::x (const x&).
Let’s take an example to illustrate this:
class Complex
{
    int real, img;
    public:
    Complex (int, int);
    Complex (const Complex& source); //copy constructor
};
Complex:: Complex (const Complex& source)
{
    this.real = source.real;
    this.img = source.img;
}
main ()
{
    Complex a (2, 3);
    Complex b = a; // this invokes the copy constructor
}
A copy constructor is called whenever an object is passed by value, returned by value or explicitly copied.

Psudocode :
Step 1: Define class STRING
    With data members : char *str;
    Constructors : STRING(char *s);
    STRING(STRING &s);
Overload operator + : STRING operator+( STRING s2);
Overload operator << : friend ostream & operator<<(ostream &print, STRING s)

Step 2: logic: assign
    STRING s1 = “VTU”;
    STRING s2 = “BELAGAUM”;
    STRING s3 = s1 + s2;

7. Design, develop, and execute a program in C++ to create a class called STACK using an array of integers and to implement the following operations by overloading the operators + and - :
   i. s1=s1 + element; where s1 is an object of the class STACK and element is an integer to be pushed on to top of the stack.
   ii. s1=s1- ; where s1 is an object of the class STACK and operator pops off the top element.
Handle the STACK Empty and STACK Full conditions. Also display the contents of the stack after each operation, by overloading the operator <<.

#include<iostream.h>

#include<conio.h>
#include<stdlib.h>
#define MAX 100

class stack
{
    int a[MAX],top;

    public: stack()
    {
        top=-1;
    }

    stack operator+(int ele);
    stack operator--(int);

    friend ostream& operator<<(ostream&,stack&);
};

stack stack::operator+(int ele)
{
    if(top==MAX-1)
        cout<<"n STACK FULL."
    else
        a[++top]=ele;
    return(*this);
}

stack stack::operator--(int)
{
    if(top==-1)
        cout<<"n STACK EMPTY."
    else
        cout<<"n The deleted element ="<<a[top--];
    return(*this);
}
ostream& operator<<(ostream &c, stack &s)
{
    if(s.top == -1)
        c << "STACK EMPTY";
    else
    
        for(int j = s.top; j >= 0; j--)
            {
                c << "|";
                c.width(4);
                c << s.a[j] << " |";
            }
        return c;
}
}  void main()
{
    stack s;
    int ch, ele;
    clrscr();
    while(ch != 4)
    
        {
            cout << "Enter the choice of operation";
            cout << "1. Push
        2. Pop
        3. Display
        4. Exit:";
            cin >> ch;
            switch(ch)
            
                {
case 1: cout<<"\n Enter the element :";
    cin>>ele;
    s=s+ele;
    break;
    case 2: s=s--;
    break;
    case 3: cout<<s;
    break;
    case 4: exit(0);
    
    }
    }
    getch();
}

/* OUTPUT
Enter the choice of operation
1. Push
2. Pop
3. Display
4. Exit: 1

Enter the element :12

Enter the choice of operation
1. Push
2. Pop
3. Display
4. Exit:1

Enter the element :23

Enter the choice of operation
1. Push
2. Pop
3. Display
4. Exit:1

Enter the element :34

Enter the choice of operation
1. Push
2. Pop
3. Display
4. Exit:1

Enter the element :56

Enter the choice of operation
1. Push
2. Pop
3. Display
4. Exit:3
Enter the choice of operation
1. Push
2. Pop
3. Display
4. Exit:2

The deleted element = 56
Enter the choice of operation
1. Push
2. Pop
3. Display
4. Exit:2

The deleted element = 34
Enter the choice of operation
1. Push
2. Pop
3. Display
4. Exit:3

| 23 |
| 12 |
Enter the choice of operation
1. Push
2. Pop
3. Display
4. Exit:2

The deleted element =23

Enter the choice of operation
1. Push
2. Pop
3. Display
4. Exit:3

| 12 |

Enter the choice of operation
1. Push
2. Pop
3. Display
4. Exit:2

The deleted element =12

Enter the choice of operation
1. Push
2. Pop
3. Display
4. Exit:2
8. Design, develop, and execute a program in C++ to create a class called LIST (linked list) with member functions to insert an element at the front of the list as well as to delete an element from the front of the list. Demonstrate all the functions after creating a list object.

```cpp
#include<iostream.h>
#include<conio.h>
#include<stdlib.h>

class node
{
    public: int info;
            class node*link;
};

class list
{
    node *head;
 public: list() {head=NULL;}
            void insert();
            void del();
            void display();
            ~list() {delete head;}
};

void list::insert()
{
    node *temp;
    temp=new node;
```
cout<<"\n Enter the element to be inserted:"
; 
cin>>temp->info;
temp->link=head;
head=temp;

void list::del()
{
    node *temp;
    if(head==NULL)
        cout<<"\n The list is empty"
; 
    else
    {
        temp=head;
        cout<<"\n The deleted element = "<<temp->info;
        head=head->link;
        temp->link=NULL;
        delete(temp);
    }
}

void list::display()
{
    node *temp;
    if(head==NULL)
        cout<<"\n The list is empty"
; 
    else
    {
        cout<<"\n The elements of the list are...

          ";
    }
}
for(temp=head;temp!=NULL;temp=temp->link)
    cout<<temp->info<<"->";
    cout<<"NULL";
}
}
void main()
{
    int ch;
    list l;
    clrscr();
    while(ch!=4)
    {
        cout="\n\n\nEnter the choice of operation:"
        cout="\n1.Insert\n2.Delete\n3.Display\n4.Exit:";
        cin>>ch;
        switch(ch)
        {
            case 1: l.insert();
                break;
            case 2: l.del();
                break;
            case 3: l.display();
                break;
            case 4: exit(0);
        }
    }
    getch();
Enter the choice of operation:
1. Insert
2. Delete
3. Display
4. Exit: 1

Enter the element to be inserted: 12
Enter the choice of operation:
1. Insert
2. Delete
3. Display
4. Exit: 1

Enter the element to be inserted: 23
Enter the choice of operation:
1. Insert
2. Delete
3. Display
4. Exit: 1

Enter the element to be inserted: 34
Enter the choice of operation:
1. Insert
2. Delete
3. Display
4. Exit: 1

Enter the element to be inserted: 45
Enter the choice of operation:
1. Insert  
2. Delete  
3. Display  
4. Exit: 3  
The elements of the list are...  
45->34->23->12->NULL  
Enter the choice of operation:  
1. Insert  
2. Delete  
3. Display  
4. Exit: 2  
The deleted element = 45  
Enter the choice of operation:  
1. Insert  
2. Delete  
3. Display  
4. Exit: 2  
The deleted element = 34  
Enter the choice of operation:  
1. Insert  
2. Delete  
3. Display  
4. Exit: 3  
The elements of the list are...  
23->12->NULL  
Enter the choice of operation:  
1. Insert
2. Delete
3. Display
4. Exit: 2

The deleted element = 23
Enter the choice of operation:
1. Insert
2. Delete
3. Display
4. Exit: 3
The elements of the list are...
12->NULL
Enter the choice of operation:
1. Insert
2. Delete
3. Display
4. Exit: 2
The deleted element = 12
Enter the choice of operation:
1. Insert
2. Delete
3. Display
4. Exit: 3
The list is empty
Enter the choice of operation:
1. Insert
2. Delete
3. Display
4. Exit: 2

The list is empty

Enter the choice of operation:
1. Insert
2. Delete
3. Display
4. Exit: 4

Enter the choice of operation:
1. Insert
2. Delete
3. Display
4. Exit: 4

9. Design, develop, and execute a program in C to read a sparse matrix of integer values and to search the sparse matrix for an element specified by the user. Print the result of the search appropriately. Use the triple <row, column, value> to represent an element in the sparse matrix.

**STRUCTURE OF EACH ELEMENT**

```c
struct SparseElem
{
    int iRow;
    int iCol;
    int iVal;
};
```
1. Create an array of structures with each element having the above defined structure to hold the triple `<row, column, value>.
2. Initialise these array with -1 in each field of every element to signify that matrix is empty.
3. Read the number of non-zero elements in the matrix.
4. Input those many values by specifying the triple `<row, column, value>` for each element.
5. Now specify the key element to be searched in the matrix.
6. Traverse the array sequentially till an element having a matching value exists or till the end is reached
   1. if a matching value is found
      1. Display the row column information.
   2. else
      1. Display message saying element not found.

**10. Design, develop, and execute a program in C to create a max heap of integers by accepting one element at a time and by inserting it immediately into the heap. Use the array representation for the heap. Display the array at the end of insertion phase.**

The binary heap data structures is an array that can be viewed as a complete binary tree. Each node of the binary tree corresponds to an element of the array. The array is completely filled on all levels except possibly lowest.

![Binary Heap Diagram](image-url)

We represent heaps in level order, going from left to right. The array corresponding to the heap above is [25, 13, 17, 5, 8, 3].
The root of the tree $A[1]$ and given index $i$ of a node, the indices of its parent, left child and right child can be computed.

```plaintext
PARENT (i)  
return floor(i/2)
LEFT (i)     
return 2i
RIGHT (i)    
return 2i + 1
```

Let's try these out on a heap to make sure we believe they are correct. Take this heap,

![Diagram of a heap with numbers and indices]

which is represented by the array [20, 14, 17, 8, 6, 9, 4, 1].

We'll go from the 20 to the 6 first. The index of the 20 is 1. To find the index of the left child, we calculate $1 \times 2 = 2$. This takes us (correctly) to the 14. Now, we go right, so we calculate $2 \times 2 + 1 = 5$. This takes us (again, correctly) to the 6.

Now let's try going from the 4 to the 20. 4's index is 7. We want to go to the parent, so we calculate $7 / 2 = 3$, which takes us to the 17. Now, to get 17's parent, we calculate $3 / 2 = 1$, which takes us to the 20.

**Heap Property**

In a heap, for every node $i$ other than the root, the value of a node is greater than or equal (at most) to the value of its parent.

$$A[\text{PARENT (i)}] \geq A[i]$$

Thus, the largest element in a heap is stored at the root.
Following is an example of Heap:

\[ \begin{array}{c}
26 \\
13 & 17 \\
5 & 3 & 8
\end{array} \]

By the definition of a heap, all the tree levels are completely filled except possibly for the lowest level, which is filled from the left up to a point. Clearly a heap of height \( h \) has the minimum number of elements when it has just one node at the lowest level. The levels above the lowest level form a complete binary tree of height \( h - 1 \) and \( 2^h - 1 \) nodes. Hence the minimum number of nodes possible in a heap of height \( h \) is \( 2^h \).

Clearly a heap of height \( h \), has the maximum number of elements when its lowest level is completely filled. In this case the heap is a complete binary tree of height \( h \) and hence has \( 2^{h+1} - 1 \) nodes.

Following is not a heap, because it only has the heap property - it is not a complete binary tree. Recall that to be complete, a binary tree has to fill up all of its levels with the possible exception of the last one, which must be filled in from the left side.

\[ \begin{array}{c}
26 \\
13 & 17 \\
5 & 8 & 3
\end{array} \]

Height of a node
We define the height of a node in a tree to be a number of edges on the longest simple downward path from a node to a leaf.

**Height of a tree**

The number of edges on a simple downward path from a root to a leaf. Note that the height of a tree with \( n \) node is \( \lfloor \log n \rfloor \) which is \( \Theta(\log n) \). This implies that an \( n \)-element heap has height \( \lfloor \log n \rfloor \).

In order to show this let the height of the \( n \)-element heap be \( h \). From the bounds obtained on maximum and minimum number of elements in a heap, we get

\[
2^h \leq n \leq 2^{h+1} - 1
\]

Where \( n \) is the number of elements in a heap.

\[
2^h \leq n \leq 2^{h+1}
\]

Taking logarithms to the base 2

\[
h \leq \log n \leq h + 1
\]

It follows that \( h = \lfloor \log n \rfloor \).

We known from above that largest element resides in root, \( A[1] \). The natural question to ask is where in a heap might the smallest element resides? Consider any path from root of the tree to a leaf. Because of the heap property, as we follow that path, the elements are either decreasing or staying the same. If it happens to be the case that all elements in the heap are distinct, then the above implies that the smallest is in a leaf of the tree. It could also be that an entire subtree of the heap is the smallest element or indeed that there is only one element in the heap, which in the smallest element, so the smallest element is everywhere. Note that anything below the smallest element must equal the smallest element, so in general, only entire subtrees of the heap can contain the smallest element.
Inserting Element in the Heap

Suppose we have a heap as follows

Let's suppose we want to add a node with key 15 to the heap. First, we add the node to the tree at the next spot available at the lowest level of the tree. This is to ensure that the tree remains complete.

Let's suppose we want to add a node with key 15 to the heap. First, we add the node to the tree at the next spot available at the lowest level of the tree. This is to ensure that the tree remains complete.
Now we do the same thing again, comparing the new node to its parent. Since 14 < 15, we have to do another swap:

Now we are done, because 15 ≤ 20.

**Maintaining the Heap Property**

Heapify is a procedure for manipulating heap data structures. It is given an array \( A \) and index \( i \) into the array. The subtree rooted at the children of \( A[i] \) are heap but node \( A[i] \) itself may possibly violate the heap property i.e., \( A[i] < A[2i] \) or \( A[i] < A[2i + 1] \). The procedure 'Heapify' manipulates the tree rooted at \( A[i] \) so it becomes a heap. In other words, 'Heapify' is let the value at \( A[i] \) "float down" in a heap so that subtree rooted at index \( i \) becomes a heap.
Outline of Procedure Heapify

Heapify picks the largest child key and compare it to the parent key. If parent key is larger than heapify quits, otherwise it swaps the parent key with the largest child key. So that the parent is now becomes larger than its children.

It is important to note that swap may destroy the heap property of the subtree rooted at the largest child node. If this is the case, Heapify calls itself again using largest child node as the new root.

**Heapify** \((A, i)\)

1. \(l \leftarrow \text{left} \[i\]\)
2. \(r \leftarrow \text{right} \[i\]\)
3. if \(l \leq \text{heap-size} \[A\] \) and \(A[l] > A[i]\)
   4. then largest \(\leftarrow l\)
   5. else largest \(\leftarrow i\)
6. if \(r \leq \text{heap-size} \[A\] \) and \(A[i] > A[\text{largest}]\)
   7. then largest \(\leftarrow r\)
8. if largest \(\neq i\)
   9. then exchange \(A[i] \leftrightarrow A[\text{largest}]\)
10. Heapify \((A, \text{largest})\)

**Analysis**

If we put a value at root that is less than every value in the left and right subtree, then 'Heapify' will be called recursively until leaf is reached. To make recursive calls traverse the longest path to a leaf, choose value that make 'Heapify' always recurse on the left child. It follows the left branch when left child is greater than or equal to the right child, so putting 0 at the root and 1 at all other nodes, for example, will accomplished this task. With such values 'Heapify' will called \(h\) times, where \(h\) is the heap height so its running time will be \(\Theta(h)\) (since each call does \(\Theta(1)\) work), which is \(\Theta(\lg n)\). Since we have a case in which Heapify's running time \(\Theta(\lg n)\), its worst-case running time is \(\Omega(\lg n)\).
**Example of Heapify**
Suppose we have a complete binary tree somewhere whose subtrees are heaps. In the following complete binary tree, the subtrees of 6 are heaps:

![Binary Tree Diagram](image)

The Heapify procedure alters the heap so that the tree rooted at 6's position is a heap. Here's how it works. First, we look at the root of our tree and its two children.

![Binary Tree Diagram](image)

We then determine which of the three nodes is the greatest. If it is the root, we are done, because we have a heap. If not, we exchange the appropriate child with the root, and continue recursively down the tree. In this case, we exchange 6 and 8, and continue.

![Binary Tree Diagram](image)

Now, 7 is greater than 6, so we exchange them.
We are at the bottom of the tree, and can't continue, so we terminate.

**Building a Heap**

We can use the procedure 'Heapify' in a bottom-up fashion to convert an array $A[1 \ldots n]$ into a heap. Since the elements in the subarray $A[\lceil n/2 \rceil+1 \ldots n]$ are all leaves, the procedure BUILD_HEAP goes through the remaining nodes of the tree and runs 'Heapify' on each one. The bottom-up order of processing node guarantees that the subtree rooted at children are heap before 'Heapify' is run at their parent.

**BUILD_HEAP (A)**

1. heap-size ($A$) ← length [$A$]
2. For $i$ ← floor(length[$A$]/2) down to 1 do
3.   Heapify ($A$, $i$)

11. Write a C program to support the following operations on a doubly linked list where each node consists of integers.
   a. Create a doubly linked list by adding each node at the front.
   b. Insert a new node to the left of the node whose key value is read as an input
   c. Delete the node of a given data, if it is found, otherwise display appropriate message.
   d. Display the contents of the list.

A **doubly linked list** is a linked data structure that consists of a set of data records, each having two
special link fields that contain references to the previous and to the next record in the sequence. It can be viewed as two singly-linked lists formed from the same data items, in two opposite orders. A doubly-linked list whose nodes contain three fields: an integer value, the link to the next node, and the link to the previous node. The two links allow walking along the list in either direction with equal ease. Compared to a singly-linked list, modifying a doubly-linked list usually requires changing more pointers, but is sometimes simpler because there is no need to keep track of the address of the previous node.

• An example doubly-linked list node with one data field:

```c
struct dllnode{
    int data;
    struct dllnode *llink;
    struct dllnode *rlink;
};
```

Adding a Node

There are four steps to add a node to a doubly-linked list:

• Allocate memory for the new node.
• Determine the insertion point to be after (pCur).
• Point the new node to its successor and predecessor.
• Point the predecessor and successor to the new node.

Current node pointer (pCur) can be in one of two states:

• it can contain the address of a node (i.e. you are adding somewhere after the first node – in the middle or at the end)
• it can be NULL (i.e. you are adding either to an empty list or at the beginning of the list).

Delete a node from doubly linked list

• Deleting a node requires that we logically remove the node from the list by changing various links and then physically deleting the node from the list (i.e., return it to the heap).
• Any node in the list can be deleted. Note that if the only node in the list is to be deleted, an empty list will result. In this case the head pointer will be set to NULL.
• To logically delete a node:
  – First locate the node itself (pCur).
  – Change the predecessor’s and successor’s link fields to point each other (see example).
  – Recycle the node using the free() function.

Algorithm for adding each node at the front

1. create a new node using malloc function. It returns address of the node to temp.
2. temp->info=info;
3. temp->llink=NULL
4. temp->rlink=NULL;
5. If first=NULL then first=temp.
6. temp->rlink=first
7. first->llink=temp; first=temp;

Algorithm for inserting a node to the left of the node
1. Create a new node using malloc function. It returns address of the node to temp.
2. temp->info=info
3. temp->llink=NULL
4. temp->rlink=NULL
5. Get the left node key value from user
6. if first= NULL print doubly linked list is empty.
7. if lvalue=first->info, call the function insert_front
8. start from the first node and traverse the node until the key is found/store that node address in cur
9. temp->llink=cur->llink

Algorithm for delete a node
1. set temp=first
2. if first=NULL print doubly linked list is empty.
3. Get node to be deleted from the user
4. if date=first ->info then first=temp->rlink and free the temp node, then first->llink=NULL.
5. start from the first node and traverse until delete key is found, then temp=temp->rlink
6. print the deleted node
7. (temp->rlink)->llink=temp->llink
8. (temp->llink)->rlink=temp->rlink

Algorithm for display
1. set temp=first
2. if first=NULL print list is empty
3. while(temp!=NULL) print temp->info and then temp=temp->rlink

12. Write a C++ program to create a class called DATE. Accept two valid dates of the form dd/mm/yy. Implement the following operations by overloading the operators + and -.
After each operation, display the result by overloading the << operator.

(i) no_of_days=d1-d2, where d1 and d2 are date objects,

   d1>=d2 and no_of_days is an integer

(ii) d2=d1+no_of_days, where d1 is a DATE object and

   no_of_dys is an integer.
#include<iostream.h>
#include<conio.h>

class date
{
    int dd,mm,yy;
    int a[13];
    long double dateno;
    void getno();
    public:
    date()
    {
        a[2]=28;
        dd=mm=yy=1;
    }
    void getdate();
    friend ostream& operator<<(ostream&,date&);
    long double operator-(date&);
    date operator+(long double);
};

void date::getdate()
{
    cout<<"Enter date"
    cout<<"Enter day(dd)";
    cin>>dd;
}
cout<<"\n\t month(mm):";
cin>>mm;
cout<<"\n\t year(yy):";
cin>>yy;
while((yy%4!=0&&dd>a[mm])||(yy%4==0 && mm==2 && dd>29)
|| (dd<=0) || mm>12 || mm<=0 )
{
    cout<<"\n Invalid entry";
    getdate();
}
getno();

void date::getno()
{
    int m=1;
dateno=(long double)yy*365+yy/4;
    if(yy%4>0)
dateno++;;
    while(m!=mm)
    {
        dateno+=a[m];
        if(yy%4==0 && m==2)
dateno++;;
        m++;;
    }
dateno+=dd;
ostream& operator<<(ostream &out, date &d1)
{
    out<<d1.dd<<"/"<<d1.mm<<"/"<<d1.yy;
    return out;
}

long double date::operator-(date &b)
{
    return(dateno-b.dateno);
}

date date::operator+(long double b)
{
    for(int i=1;i<=b;i++)
    {
        dd++;
        if(dd>a[mm])
        {
            mm++;
            dd=1;
        }
        if(mm>12)
        {
            yy++;
        }
    }
    return *this;
}
mm=1;

if(yy%4==0)
a[2]=29;

return *this;

void main()
{

date d1,d2,d3;
clrscr();
d1.getdate();
cout<<"\n\td1="<<d1;
d2.getdate();
cout<<"\n\td2="<<d2;
long double s;
s=d1-d2;
cout<<"\n\nThe difference between the two dates = ";
cout<<s;
cout<<"\n\nEnter the no. of days to be added to the date "<<d1<<" :";
cin>>s;
d3=d1+s;
cout<<"\n\nNew date is..."<<d3;
getch();
}
/* OUTPUT */

RUN 1:

Enter date
  day(dd): 1
  month(mm): 1
  year(yy): 2001
  d1=1/1/2001

Enter date
  day(dd): 1
  month(mm): 1
  year(yy): 2000
  d2=1/1/2000

The difference between the two dates = 366

Enter the no. of days to be added to the date 1/1/2001: 365

New date is... 1/1/2002

RUN 2:

Enter date
  day(dd): 29
  month(mm): 2
  year(yy): 2001

Invalid entry

Enter date
  day(dd): 29
  month(mm): 2
  year(yy): 2000
  d1=29/2/2000

Enter date
day(dd): 28
month(mm): 2
year(yy): 2000
d2 = 28/2/2000

The difference between the two dates = 1

Enter the no. of days to be added to the date 29/2/2000 : 365

New date is... 28/2/2001

Enter date
day(dd): 1
month(mm): 1
year(yy): 2000
d1 = 1/1/2000

Enter date
day(dd): 1
month(mm): 1
year(yy): 1999
d2 = 1/1/1999

The difference between the two dates = 365

Enter the no. of days to be added to the date 1/1/2000 : 365

New date is... 31/12/2000
13. Write a C++ program to create a class called OCTAL which has the characteristics of an octal number. Implement the following operations by writing an appropriate constructor and an overloaded operator +.

(i) OCTAL h = x; where x is an integer.
(ii) int y = h + k; where h is an OCTAL object and k is an integer

Display the OCTAL result by overloading the operator <<. Also display the values of h and y.

**Implementation**

```cpp
#include <iostream.h>
#include <conio.h>
#include <math.h>

class octal
{
private:
int o;
public:
  octal();
  octal(int);
  ~octal();
  int dectooct(int x);
  int octtodec(int x);
friend ostream &operator<<(ostream &,octal);
int operator +(int);
};
octal::octal()
{
}
octal::octal(int x)
{
o=dectooct(x);
}
octal::~octal()
{
}
int octal::dectooct(int x)
{
int i=0,sum=0,rem;
while(x!=0)
{
rem=x%8;
sum=sum+rem*pow(10,i);
i++;
x=x/8;
}
return sum;
```
```c++
int octal::octtodec(int x)
{
    int i=0, sum=0, rem;
    while(x!=0)
    {
        rem = x % 10;
        sum = sum + rem * pow(8, i);
        i++;
        x = x / 10;
    }
    return sum;
}

ostream &operator<<(ostream &print, octal x)
{
    print << x.o;
    return print;
}

int octal::operator+(int x)
{
    return octtodec(o) + x;
}

main()
{
    clrscr();
    int x, y, k;
    cout << endl << "Enter the value of x in decimal notation: ";
    cin >> x;
    octal h(x);
    cout << endl << "Corresponding value of x in octal notation, h = " << h;
    cout << endl << "Enter the value of k in decimal notation: ";
    cin >> k;
    cout << "The value of k = " << k;
    y = h + k;
    cout << endl << "The value of h+k in decimal notation, y = " << y;
    getch();
    return 0;
}
```

// INPUT & OUTPUT
// Enter the value of x in decimal notation: 10
// Corresponding value of x in octal notation, h = 12
// Enter the value of k in decimal notation: 10
// The value of k = 10
// The value of h+k in decimal notation, y = 20
14. Design, develop, and execute a program in C++ to create a class called BIN_TREE that represents a Binary Tree, with member functions to perform inorder, preorder and postorder traversals. Create a BIN_TREE object and demonstrate the traversals.

A binary tree is made of nodes, where each node contains a "left" pointer, a "right" pointer, and a data element. The "root" pointer points to the topmost node in the tree. The left and right pointers recursively point to smaller "subtrees" on either side. A null pointer represents a binary tree with no elements -- the empty tree. The formal recursive definition is: a **binary tree** is either empty (represented by a null pointer), or is made of a single node, where the left and right pointers (recursive definition ahead) each point to a **binary tree**.

Step 1: define **class node with following data members**:  

```cpp
class node {  
  node* left;  
  int info;  
  node* right;  
};
```

Step 2: define **class bintree with following data members**:  

```cpp
class bintree {  
  node* root;  
};
```

Member functions:

```cpp
bintree(int num) {  
  root=NULL;  
}
```

```cpp
node* createtree();  
void intrav(node*);  
void pretrav(node*);  
void posttrav(node*);  
```