Radix Sort:

Consider we want to sort \( n \) integers in the range \( 0 \) to \( n^c - 1 \) (\( c \) is a constant). If we directly use Binsort to sort these numbers, we will need \( n^c \) operations (expensive in terms of time & memory). Hence we decompose these numbers in terms of a "radix" \( r \). Most commonly, an integer is decomposed using a radix \( 10 \). Eg: \( 1204 = 4 \times 10^0 + 0 \times 10^1 + 2 \times 10^2 + 1 \times 10^3 \).

The most significant digit = 1.
The least "    "  = 4.

In radix sort, we first decompose the numbers using some radix \( r \), and then use a binsort on each individual digit. We start from the least significant digit and move towards the most significant digit.

Eg: Consider the foll. set of numbers.
961, 243, 521, 23, 9, 475, 675, 800, 917, 632, 333

Bin-sort by last digit

<table>
<thead>
<tr>
<th>Digit</th>
<th>Binsort Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>800, 809</td>
</tr>
<tr>
<td>1</td>
<td>961, 521</td>
</tr>
<tr>
<td>2</td>
<td>632</td>
</tr>
<tr>
<td>3</td>
<td>243, 23, 333</td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>475, 675</td>
</tr>
<tr>
<td>6</td>
<td>917</td>
</tr>
<tr>
<td>7</td>
<td>961</td>
</tr>
<tr>
<td>8</td>
<td>475, 675</td>
</tr>
<tr>
<td>9</td>
<td></td>
</tr>
</tbody>
</table>

Bin-sort by last but one digit
Start from the list of numbers sorted by $d^{th}$ most significant digit. When you sort this list by the $(d-1)^{th}$ most sig. digit, you get a list sorted by all digits from the $(d-1)^{th}$ most sig. digit to the least sig. digit. The $(d-1)$ in the list sorted by $(d-1)^{th}$ most sig. digit, two numbers with the same digit in the $(d-1)^{th}$ place will remain sorted by the $d^{th}$ most sig. digit. (Why?)

The linked list representation is useful for radix sort just as it was useful for bin sort. This is because multiple numbers could fall into a bin.
"m" numbers from 0 to 1000. A radix of 1000 will require creating a table of size 1000, followed by bin sort. Time taken is 1000 + 3m operations.

A radix of 10 will require creating a table of size 10 followed by 3 bin sorts— one on each of the three digits. Time taken = 10 + 3m operations. If the range of numbers is very large (say 0 to 10^10) it is inefficient to pick a very large radix.

Given integer \( x \), you extract least sig digit using \( x \div 10 \), second one using \( \lfloor (x \div 100) / 10 \rfloor \), third one using \( \lfloor (x \div 1000) / 10 \rfloor \) etc.

This was for radix \( r = 10 \).

For general \( r \), the operations are \( x \div r \), \( (x \div r^2) / r \), \( (x \div r^3) / r \), etc.
Stack:
A stack is a last-in-first-out (LIFO) data structure. We can add a data element/object at one end of the stack. Elements of the stack can be deleted also from the same end. This location where elements are added or from where elements are deleted is called the Top of the Stack.

The stack can be implemented using arrays or linked lists. The delete operation is usually called "pop". The add operation is called "push".

Array - Based Implementation.

```cpp
template <class T>

class Stack {

private:
int top, MaxSize;
T * sa;  // stack array

public:
Stack (int maxsize) {
    sa = new T[maxsize];
    top = -1;
}

...}
```
getTop() { return top; }

getTopElement() { if (is_empty())
    return sa[top];
  else return NULL;

bool is_empty() { return (top == -1); }

bool is_full() { return (top == maxsize); }

void push(T& x)
{
  if (top < maxsize - 1)
  {
    top++;
    sa[top] = x;
  }
}

T pop() { top--; return sa[top - 1]; }

dist-based representation

template <class T>
class Stack_LinkedList : private Chain <T>
{
  public:  
    is_empty()
    {
      return Chain<T>::is_empty();
    }

    T getTop()
    {
      T pop()
      {
        if (is_empty()) return NULL;
        Node<T>* q = head->next;
        head->next = q->next;
      }
}
\[ T \ y = q \rightarrow \text{data}; \]
\[ \text{delete } q; \]
\[ \text{return } y; \]

```c
void push (T y) {
    Node<T> *q = new Node<T> (*p;
    q \rightarrow \text{data} = y; q \rightarrow \text{next} = \text{NULL};
    p = head \rightarrow \text{next};
    head \rightarrow \text{next} = q \rightarrow i;
    q \rightarrow \text{next} = p \rightarrow \text{next};
}
```

Note: In our implementation, we keep a header node. The top of the stack is given by the node head \rightarrow next. This facilitates efficient implementation. If the node head \rightarrow next were the bottom of the stack, all operations would be very expensive (unless we separately kept track of the top of the stack). Note that a linked-list stack implementation typically has no "IsFull" operation (unless machine memory is filled up).
Applications of a stack

PARANThESIS MATCHING

Consider an expression:
\((a \ast (b + c) + d)\)

1 2 3 4 5 6 7 8 9 10 11

The parentheses at positions 1 and 11 match each other.

In an expression like \(a + b \ast (c + d)\), the last parenthesis has no matching left parenthesis.

 Aim: To write an algorithm to take an input expression, and output the matching parentheses, including declaring those with no match.

Algorithm:
1) Scan input expression from left to right.
2) Push any \( ( \) symbol onto the stack.
3) If you encounter a \( ) \) symbol, print its position in the input string. If the stack is empty, declare "No match"; otherwise pop a \( ( \) from the stack and print stack-top + 1 as the matching position.
In this game, we have three towers 1, 2, 3. A number of disks (n) can be stored in any tower with the constraint that a larger disk can never be placed on top of a smaller disk. The disks are all of distinct sizes. The disks can be moved only one at a time and must be placed only in one of the three towers. Initially, all n disks are present in tower 1, with the largest disk at the bottom and smallest one at the top. The aim is to move all n disks from T1 to T2 using T3 as intermediate storage.

Example: n = 2

```
T1   T2   T3
1     2   *
```

```
T1   T2   T3
1     2   1
```

```
T1   T2   T3
1   2
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```

```
T1   T2   T3
```
Algorithmic solution (using recursion)

To move $n$ disks from $T_1$ to $T_2$ (using $T_3$).

→ Move $(n-1)$ disks (excluding largest one) from $T_1$ to $T_3$ (using $T_2$).

→ Move largest disk from $T_1$ to $T_2$.

→ Move $(n-1)$ disks from $T_3$ to $T_1$ (using $T_2$).

```c
void TowersOfHanoi (int n, int T1, int T2, int T3)
{   // to move n disks from T1 to T2 using T3 as intermediate storage
    if (n > 0)
    {
        TowersOfHanoi (n-1, T1, T3, T2);
        printf("Move top disk %d from T%d to T%d", n, T1, T2);
        TowersOfHanoi (n-1, T3, T2, T1);
    }
}
```
Each of the towers can be represented as a stack, either with array-based or linked-list-based representation.
The conventional way of writing arithmetic expressions (e.g. \( a + b \times c + d \)) is called infix. Another form called prefix is more appropriate for expression evaluation. In the prefix form, the operators are written before the operands.

\[
\begin{align*}
a + b \times c &= + b c a \\
(a + b) \times c &= \ast + a b c \\
a + (b \times c) &= + b c a \\
a + b \times c - d &= - + b c a d
\end{align*}
\]

A prefix expression can be evaluated using a stack, using following algorithm:

1. reverse prefix expression
2. scan reversed string from beginning to end.
   - Push any operand onto the stack.
   - If you encounter an operation \( \text{op} \), pop two elements \( y_2, y_1 \) from the stack. Push \( y_1 \text{ op } y_2 \) onto the stack.
3. return value on the top of the stack as the final result.
Example: \(3 + (7 - 5) \times 8 + 6\)

Reversed string = \(6 3 8 5 7 - \times + +\)

Stack

\[
\begin{align*}
& 6 \\
& 6 3 8 5 7 - \times + + \\
& 6 3 8 5 7 - \times + + \\
& 6 3 8 5 7 - \times + + \\
& 6 3 8 5 7 - \times + + \quad \text{eval \((7-5)\), push onto stack}
\end{align*}
\]

\[
\begin{align*}
& 6 3 8 2 \\
& 6 3 8 2 \\
& 6 3 16 \\
& 6 19 \\
& 25 \quad \text{eval \(19+6\), push} \\
& \quad \text{pop final result.}
\end{align*}
\]
Algorithm to Convert Infix to Prefix

Notation:

Input: Infix expression

for (i = 1 to length (infix exp); i++)

if (infix[i] == '(') push onto Operator stack;
elseif (infix[i] is an operand) push onto Operand stack;
elseif (infix[i] == ')')

repeat steps below till you encounter a ']' on the operator stack

-> OP = pop from operator stack
-> a1, a2 = pop twice from operand stack
-> push "OP a1 a2" onto operand stack

elseif (infix[i] is an operator)

repeat steps below till a lower precedence operator (as compared to infix[i]) is popped from the stack
→ OP = pop from operator stack
→ a1, a2 = pop twice from operand stack
→ push "OP a1 a2" onto operand stack
If the stack operator stack has lower precedence than infix [i] 

→ push infix [i] onto operator stack

At the end of the input, repeat foll. steps till operator stack is empty

→ op = pop from operator stack
→ a1, a2 = pop twice from operand stack
→ push "OP a1 a2" onto operand stack

Final expression = pop from operand stack

eg:  A + B * C + D