Implementations :

- Stacks using arrays
- Stacks using Linked List
- Infix->Postfix
- Infix->Prefix
- Recursion Applications :
  - factorial
  - tower of hanoi
  - Fibonacci
- Singly Linked List
- Doubly Linked List
- Circular Linked LIst
- Circular Doubly Linked List

Stack

- All insertions and deletions of entries are at one end(`Top of stack`)
- Operations:
  - `push`
  - `pop`
  - `peep`
  - `empty`: checks if empty
  - `overflow`: checks if full
- `LIFO` Principle.

```c
// Code to convert infix expression to postfix
#include <stdio.h>
#include <stdlib.h>
#define SIZE 10
int inp_prec(char ch)
  {
    switch(ch)
    {
      case '+':
      case '-':return 1;
      case '*':
      case '/':return 3;
      case '$':return 6;
      case '(':return 9;
      case ')':return 0;
      default:return 7;
    }
 }
  int stk_prec(char ch)
  {
    switch(ch)
    {
      case '+':
      case '-':return 2;
      case '*':
      case '/':return 4;
      case '$':return 5;
      case '(':return 0;
      case '#':return -1;
      default:return 8;
   }
}

char peep(char * s, int t)
 {
   return s[t];
 }

 void push(char *s, int *t, char x)
 {
   ++*t;
   s[*t]=x;
 }

  char pop(char *s, int *t)
  {
    char x;
     x=s[*t];
    --*t;
   return x;
```

```
  }


void convert(char* in,char* post){
    int* top = malloc(sizeof(int));
    char stk[SIZE];
    int i=0;
    int j=0;
    *top= -1;
    char ch ;
    push(stk,top,'#');
    while(in[i]!='\0')
    {
        ch = in[i];
        while (stk_prec(peep(stk,*top))>inp_prec(ch)) {
            post[j++] = pop(stk,top);
        }
        if (stk_prec(peep(stk,*top))!=inp_prec(ch)) {
            push(stk,top,ch);
        }
        else{
            pop(stk,top);
        }
        i++;
    }
    while (peep(stk,*top)!='#') {
        post[j++] = pop(stk,top);
    }
    post[j] = '\0';
}
int main()
{
  char infix[10],postfix[10];
  printf("\nType Infix\n");
  scanf("%s",infix);
  convert(infix,postfix);
  printf("\nPostfix : %s\n",postfix);
}
```

```
// Code to evaluate a postfix expression
#include<stdio.h>
#include<string.h>
#include<ctype.h>
#include<math.h>
int compute(char symb,int op1,int op2)
{
        switch(symb)
        {
                case '+':return(op1+op2);
                case '-':return(op1-op2);
                case '*':return(op1*op2);
                case '/':return(op1/op2);
                case '%':return(op1%op2);
                case '$':
                case '^':return(pow(op1,op2));
        }
}
int main()
{
        char postfix[100];
        int st[100];
        int top=-1;
        int res;
        char symb;
        int op1,op2;
        printf("Enter the postfix expression\n");
        scanf("%s",postfix);
        for(int i=0;i<strlen(postfix);i++)
        {
                symb=postfix[i];
                if(isdigit(symb))
                {
                        st[++top]=symb-'0';
                }
                else{
                        op2=st[top--];
                        op1=st[top--];
                        res=compute(symb,op1,op2);
                        st[++top]=res;
                }
        }
        res=st[top];
        printf("result=%d\n",res);
```

```
        return(0);
}
```

- For Infix to Prefix,
- reverse the infix string,
- swap all `(` and `)`.
- Run this through postfix algorithm
- reverse the result again.

Data Structures

- Scheme of organising data for efficient computation
- Way the data is organised in memory.

> **Abstract Data Type** : used to represent the operations associated with an entity from the POV of the user irrespective of implementation.

Memory Allocation methods

- Static
  - Allocated by the compiler
  - Size predetermined at compile time
  - Memory Allocated in the stack
  - Disadvantages:
    - No mem alloc at runtime
    - Under/Overutilization of memory
    - Memory not deleted, its content is overwritten
- Dynamic
  - Obtaining and releasing memory at runtime
  - Low level functions
  - `stdlib.h` defines the functions in the C language
    - `malloc(size_t size)`
      - `size` number of bytes
      - returns `void*` pointer
      - `NULL` on failure.also what if this i get
    - `calloc(size_t number,size_t size)`
      - `number` : number of items
      - `size` : size of each item
      - `void*` if allocated, `NULL` if failure
    - `realloc(ptr,size)`
      - reallocated the size of already existing block
    - `free(ptr)`
      - releases the memory into the heap

ArrayList	Linked list
Fixed size: Resizing is expensive	Dynamic size
Insertions and Deletions are inefficient	Insertions and Deletions are efficient
Elements in contiguous memory locations	Elements not in contiguous memory locations
May result in memory wasteage if all the allocated space is not used	Since memory is allocated dynamically(as per requirement) there is no wastage of memory.
Sequential and random access is faster	Sequential and random access is slow

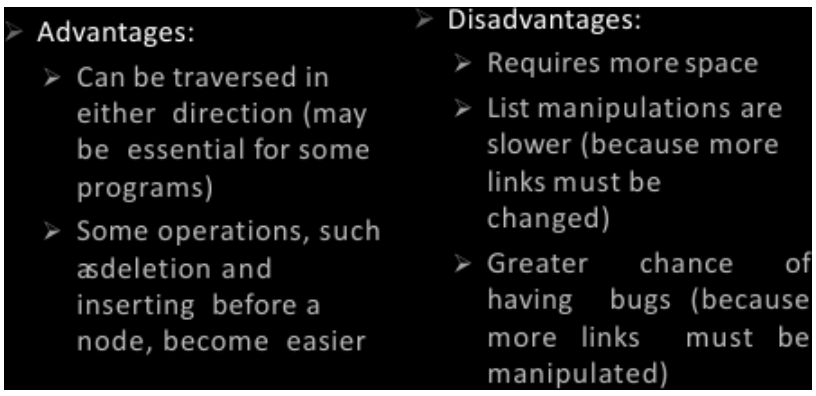
- In arraylist , memory is placed sequentially and accessed sequentially
- In LinkedList , memory is places randomly and accessed sequentially

Linked List

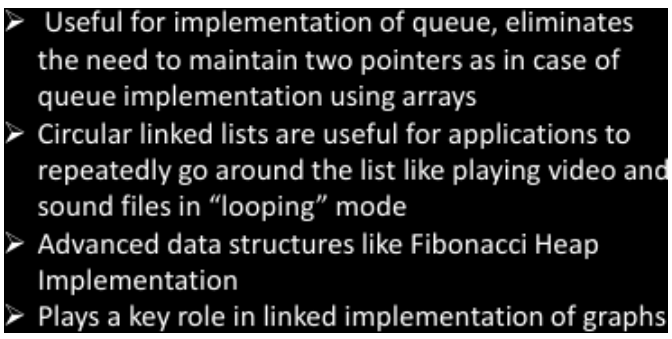
- Methods:
  - InsertFront
  - InsertRear
  - InsertAtPos
  - DeleteFront
  - DeleteRear
  - DeleteAtPos
  - Display

    - Search
- Operations
    - Remove Duplicates
    - Reverse a linked list
    - Reverse using recursion
    - Pairs of nodes with a given sum

Comparing Doubly to Singly linked list:

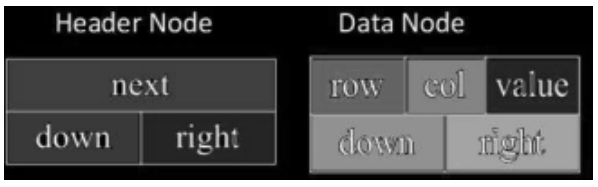


Circular Linked List



Sparse Matrix

- In a 2D array(Matrix) if most elements are zero its called a sparse matrix.
- This wastes a lot of space if stored normally.
- We use alternative ways:
    - Triple notation
    - Linked notation
- Triple Notation
    - Represents data as a tuple `<rowno columnno Value>`
    - First row holds `<total rows,total cols,total values>`
    - Operations:
        - `createSparseMatrix()`
        - `transpose()`
        - `AddSparseMatrix()`
        - `multiply()`
- Linked Notation
    - 2 Types of Nodes:



Skip Lists

- Normal lists have the following time complexities :
    - size : `O(n)`
    - isEmpty : `O(1)`
    - Search : `O(n)`
    - Insertion and deletion : `O(1)`
- Skip list :
    - `O(log(n))` for insert,delete and search
    - Gets the best features of arrays(like searching) while maintaining a structure similar to a linked list
    - This is made possible by constructing a heirarchial structure with each lower layer containing more elements than the previous one.
    - Search Algo:
        - Start searching the sparsest level
        - Continue till we find 2 consecutive elements , one greater and one smaller than the desired key.
        - We continue this search in lower levels until the element is found.
    - The lowest level (0) is a sorted linked list
    - The level 1 is the same list with only alternate elements
    - Level 2 is every fourth element and so on.

- $log_2(n)$ levels
- But for insertion into this list we would have to change the positioning of every node which would again take $O(n)$ time .
- Search Algo:
  - To search x in list :
    - we compare x to the element right after current position
    - if $x == y$ then return y
    - if $x > y$ move right
    - if $x < y$ drop down
  - if drop down returned after lowest level it means element doesn't exist in the list
- Insertion :
  - Use of *randomization* .
  - We insert into the bottom list and then flip a coin. If it returns heads we insert into the next level otherwise we stop.
  - Worst case time complexity is when all tosses give heads
- Deletion :
  - First search for he given key
  - If it is found then we delete all the positions above that.

Paranthesis Matching

- Algorithm :
  - Read the input string.
  - For every `(/{/[` push the charachter onto the stack.
  - have 3 separate cases for `),}&]` where you pop from the stack and return 1 *only* if the popped paranthesis matches with the one encountered in input.