This documentation is for reference purpose only and is for those who have attended the classroom sessions at Thinking Machines

- **During your classroom session appropriate theory needs to be written against each example.**
- **You are required to bring this documentation daily for your classroom sessions.**
- **Some examples won't compile. They have been written to explain some rules.**
- **If you try to understand the examples without attending theory sessions then may god help you.**

Index

index continued.....

Data structures.

The topic of data structures can be explained as concept/science related to organization and management of data used by an application.

For processing data needs to be organized in a a proper form. This form is defined as a data structure. Some data structures require more or less space than others. Depending upon the data structure used, some process takes more time or some process takes less time. We need to make proper choice to save time/space while designing our applications.

Array, stack, queue, linked list etc. are examples of data structures.

UNIT-I
Introduction: Basic Terminology, **Data types & its classification,** Algorithm complexity notations likebig O,**Array Definition, Representation and Analysis of Arrays, Single and Multidimensional Arrays**, **Address calculation, Array as Parameters, Ordered List and operations**,**Sparse Matrices**,Storage pools,Garbage collection. **Recursion-definition and processes, simulating recursion,Backtracking, Recursive algorithms, Tail recursion, Removal of recursion. Tower of Hanoi Problem.**

UNIT II
Stack:**Array Implementation of stack, Linked Representation of Stack**, **Application of stack: Conversion of Infix to Prefix and Postfix Expressions and Expression evaluation**, **Queue, Array and linked implementation of queues**, **Circular queues, D-queues and Priority Queues. Linked list,Implementation of Singly Linked List**, **Two-way Header List, Doubly linked list, Linked List in Array**, Generalized linked list, Application: Garbage collection and compaction, **Polynomial Arithmetic.**

UNIT III
Trees:Basic terminology, Binary Trees,algebraic Expressions,Complete Binary Tree, Extended Binary Trees, Array & Linked Representation of Binary trees, Traversing Binary trees, Threaded Binary trees, **Binary Search Tree (BST)**, AVL Trees, B-trees. Application: Algebraic Expression, Huffman coding Algorithm.

UNIT IV
Internal and External sorting:Insertion Sort,Bubble Sort,selection sort,Quick Sort,Merge Sort, Heap Sort, Radix sort, Searching & Hashing: Sequential search, binary search, Hash Table,Hash Functions, Collision Resolution Strategies, Hash Table Implementation. Symbol Table, Static tree table, Dynamic Tree table.

UNIT-V
Graphs:Introduction,Sequential Representations of Graphs,Adjacency Matrices,Traversal, Connected Component and Spanning Trees, Minimum Cost Spanning Trees.

Data types & its classification

Simple / Primitive data types : If the data is (scalar) single valued, then it can be organized using simple / primitive data types.

Example : roll number can be stored as an int type value, or information about gender can be stored as char type value. Hence long,int,short,double,float and char are simple/premitive data types.

Complex data types : If the data has several attributes (composite) then it is organized using complex data type.

Example :

```
struct Student
{
int rollNumber;
char name[21];
char sex;
int age;
};

struct Student g;

g.rollNumber=101;
strcpy(g.name,"Anil");
g.sex='M';
g.age=20;
```

data type of structure (g) is (struct Student). The data type (struct Student) has several values against its different attributes, hence the data type (struct Student) is a complex data type.

In C++ same is true for a class, what is true for a struct in C.

An abstract data type (**ADT**) is an which is generic and is independent of value/implementation details. For example int,long,short are ADT as it is not clear as what will be the value as int or long or short.

```
int rollNumber;
```

the information about the value of rollNumber is not clear/visible hence the data type of rollNumber which is (int) is said to be an abstract data type.

****** Note : Don't relate the ADT with the concept of abstract class in OOPL. The term abstract in ADT means that the implementation/value is not specified/clear.

Array Definition :

An array is a data structure which holds multiple values of same types.

Example :

int x[10]; (x) is an array of 10 elements. Every element is of type int.

struct Student s[100]; (s) is an array of 100 elements. Every element is a structure of type struct Student.

Bulb g[100]; (g) is an array of 100 elements. Every element is an object of type Bulb. Bulb is a class.

Array Definition, Representation and Analysis of Arrays, Single and Multidimensional Arrays (Discussed in the classroom session)

Row major : int x[10][3]; 10 rows and 3 columns
Column Major : int x[20][6]; 20 columns and 6 rows.

Address calculation :

One dimensional array : Base address + (index * size of one element)

Two dimensional array :

Row major

Address of a[i][j]th element = BA + [N * (i - LBR) + (j - LBC)] * W

Where BA = Base address
W = Number of bytes occupied by each element
N = Number of columns
LBR = lower bound of row
LBC = lower bound of column.

Column major

Address of a[i][j]th element = BA + [(i - LBR) + M * (j - LBC)] * W

Where BA = Base address
W = Number of bytes occupied by each element
M = Number of rows
LBR = lower bound of row
LBC = lower bound of column.

The calculations have been discussed in the classroom sessions in details.

Array as Parameters :

```
#include<stdio.h>
int computeSum(int *a,int size)
{
int e,t;
for(e=0,t=0;e<size;e++)
{
t=t+a[e];
}
return t;
}
int main()
{
int x[10],y,sum;
for(y=0;y<=9;y++)
{
printf("Enter a number");
scanf("%d",&x[y]);
}
sum=computeSum(x,10);
printf("Total is %d\n",sum);
return 0;
}
```

In the above program the computeSum function can also be defined in the following fashion.

```
int computeSum(int a[],int size)
{
int e,t;
for(e=0,t=0;e<size;e++)
{
t=t+a[e];
}
return t;
}
int main()
{
// same as previous example
}
```

Ordered list :

An ordered list is a list in which the order of the items is significant. The items in an ordered list are not necessarily sorted.

int x[10];

10	2	4	21	30	55	2	23	33	44
0	1	2	3	4	5	6	7	8	8

The above defined array represents an ordered list with index as 0 to 9. The contents of the ordered list are not sorted.

Various operations like find, remove, add, insert can be designed for the above ordered list.

We will be looking into some other versions of ordered list like linked list, stack, queue, hast table, binary tree later on.

Sorted list :

An sorted list is a list in which the contents of list are in sorted form. Array, linked list, trees etc. (if contains data in sorted form) can be looked upon as sorted lists.

Sparse Matrices :

A sparse matrix is a matrix which as more zero valued elements then non zero valued elements.

Let us write a program to determine whether the matrix is sparse matrix or not.

```
#include<stdio.h>
int main()
{
int x[5][5],r,c,z,zeroValuedCellCount,rows,columns;
rows=5;
columns=5;
zeroValuedCellCount=0;
for(r=0;r<rows;r++)
{
for(c=0;c<columns;c++)
{
printf("Enter value for %d row index and %d column index",r,c);
scanf("%d",&x[r][c]);
if(x[r][c]==0)
{
zeroValuedCellCount++;
}
}
}
printf("Matrix\n");
```

```
for(r=0;r<rows;r++)
{
for(c=0;c<columns;c++)
{
printf("%5d",x[r][c]);
}
printf("\n");
}
if(zeroValuedCellCount>(rows*columns)/2)
{
printf("It is a sparse matrix");
}
else
{
printf("It is not a sparse matrix");
}
return 0;
}
```

Now let us write a program that will keep the information of sparse matrix in another array with enough information to generate the sparse matrix whenever required.

```
#include<stdio.h>
#include<malloc.h>
void acceptData();
void printData();
int **data;
int rows,columns,zeroValuedCellCount;
int dataFeeded;
int isSparseMatrix;
int main()
{
int ch;
rows=5;
columns=5;
dataFeeded=0;
data=NULL;
while(1)
{
printf("1 Feed 5x5 matrix data\n");
printf("2. Print 5x5 matrix data\n");
printf("3. Exit \n");
printf("Enter your choice : ");
scanf("%d",&ch);
if(ch==1)
{
acceptData();
}
if(ch==2)
{
```

```
if(dataFeeded==0)
{
printf("Data for matrix not feeded\n");
continue;
}
printData();
}
if(ch==3)
{
break;
}
}
return 0;
}
void acceptData()
{
int x[5][5],r,c,nonZeroValuedCellCount,i;
rows=5;
columns=5;
zeroValuedCellCount=0;
for(r=0;r<rows;r++)
{
for(c=0;c<columns;c++)
{
printf("Enter data of %d row index and %d column index : ",r,c);
scanf("%d",&x[r][c]);
if(x[r][c]==0)
{
zeroValuedCellCount++;
}
}
}
printf("The feeded matrix is \n");
for(r=0;r<rows;r++)
{
for(c=0;c<columns;c++)
{
printf("%7d ",x[r][c]);
}
printf("\n");
}
dataFeeded=1;
if(zeroValuedCellCount>(rows*columns)/2)
{
isSparseMatrix=1;
if(data!=NULL) free(data);
nonZeroValuedCellCount=(rows*columns)-zeroValuedCellCount;
data=(int **)malloc(sizeof(int *)*(nonZeroValuedCellCount+1));
i=0;
while(i<nonZeroValuedCellCount+1)
```

```
{
data[i]=(int *)malloc(sizeof(int)*3);
i++;
}
data[0][0]=rows; // number of rows in sparse matrix
data[0][1]=columns; // number of columns in sparse matrix
data[0][2]=nonZeroValuedCellCount; // number of non zero valued data elements in sparse matrix
i=1;
for(r=0;r<rows;r++)
{
for(c=0;c<columns;c++)
{
if(x[r][c]!=0)
{
data[i][0]=x[r][c]; // data kept in i(th) row of the new matrix
data[i][1]=r; // row index of the sparse matrix where the data should reside
data[i][2]=c; // column index of the sparse matrix where the data should reside
i++;
}
}
}
printf("The feeded matrix is a sparse matrix\n");
printf("The newly created matrix is \n");
for(r=0;r<nonZeroValuedCellCount+1;r++)
{
for(c=0;c<3;c++)
{
printf("%7d ",data[r][c]);
}
printf("\n");
}
printf("The size of the sparse matrix is %d\n",(sizeof(int)*rows)*(sizeof(int)*columns));
printf("The size of newly created matrix to store information of the sparse matrix is
%d\n",sizeof(int)*(nonZeroValuedCellCount+1)*3+(sizeof(int *)*(nonZeroValuedCellCount+1))
+sizeof(int **));
}
else
{
isSparseMatrix=0;
if(data==NULL)  free(data); // memory leak problem, discussed in classroom session
data=(int **)malloc(sizeof(int *)*rows);
i=0;
while(i<rows)
{
data[i]=(int *)malloc(sizeof(int)*columns);
i++;
}
for(r=0;r<rows;r++)
{
for(c=0;c<columns;c++)
```

```
{
data[r][c]=x[r][c];
}
}
printf("The feeded matrix is not a sparse matrix\n");
}
}
void printData()
{
int r,c,i;
int x[5][5];
if(isSparseMatrix)
{
for(r=0;r<rows;r++)
{
for(c=0;c<columns;c++)
{
x[r][c]=0;
}
}
i=1;
while(i<=data[0][2])
{
x[data[i][1]][data[i][2]]=data[i][0];
i++;
}
}
else
{
for(r=0;r<rows;r++)
{
for(c=0;c<columns;c++)
{
x[r][c]=data[r][c];
}
}
}
for(r=0;r<rows;r++)
{
for(c=0;c<columns;c++)
{
printf("%7d ",x[r][c]);
}
printf("\n");
}
}
```

Recursion

Definition : Recursion is a method by which you can break down a task into one or more sub taks which are similar in form to the original form.

Processes : To implement recursion a function places a call to itself. A function which has a call to itself is called a recursive function.

Direct recursion

Example 1

```
#include<stdio.h>
void lmn(int);
int main()
{
lmn(1);
return 0;
}
void lmn(int p)
{
if(p==4)
{
return;
}
printf("%d\n",p);
lmn(p+1);
}
```

Example 2

```
#include<stdio.h>
void lmn(int);
int main()
{
lmn(1);
return 0;
}
void lmn(int p)
{
if(p==4)
{
return;
}
lmn(p+1);
printf("%d\n",p);
}
```

Example 3

```
#include<stdio.h>
int lmn(int);
int main()
{
int x,y;
```

```
x=5;
y=lmn(x);
printf("%d",y);
return 0;
}
int lmn(int p)
{
int n;
if(p==1) return 1;
n=lmn(p-1)*p;
return n;
}
```

Example 4

```
#include<stdio.h>
void lmn(int,int);
int main()
{
lmn(0,1);
return 0;
}
void lmn(int p1,int p2)
{
int sum;
if(p2==55) return;
if(p1==0) printf("%d\n",p2);
sum=p1+p2;
printf("%d\n",sum);
lmn(p2,sum);
}
```

Indirect recursion

```
#include<stdio.h>
void lmn(int);
void pqr(int);
int main()
{
pqr(1);
}
void pqr(int p)
{
if(p==6) return;
lmn(p);
}
void lmn(int e)
{
printf("%d\n",e*e);
```

```
pqr(e+1);
}
```

Simulating recursion : If the language doesn't have support for recursion then it is implemented through stack to simulate recursion. We will implement it when we learn the topic of stack.

Backtracking : Discussed in class room session. The code will be implemented in Binary Search Tree.

Tail recursion : Discussed in classroom session. Aboive mentioned **Example 1** against title direct recursion is an example of tail recursion.

Removal of recursion

To remove recursion, if it is an example of tail recursion then recursion can be removed by converting it to an iterative function and if it is not an example of tail recursion then recursion can be removed using stack.

Converting tail recursion to an iterative solution.
// an example with tail recursion

```
#include<stdio.h>
void lmn(int);
int main()
{
lmn(1);
return 0;
}
void lmn(int p)
{
if(p==4)
{
return;
}
printf("%d\n",p);
lmn(p+1);
}
```

// converting it to iterative function

```
#include<stdio.h>
void lmn();
int main()
{
lmn();
return 0;
}
void lmn()
{
int p=1;
```

```
while(p<4)
{
printf("%d\n",p);
p=p+1;
}
}
// the above concept has been discussed in the classroom session.
```

Tower of Hanoi Problem.

```
#include<stdio.h>
void towerOfHanoi(int n, char LEFT, char RIGHT, char CENTER)
{
if(n>0)
{
towerOfHanoi(n-1, LEFT, CENTER, RIGHT);
printf("\n\tMove disk %d from %c to %c", n, LEFT, RIGHT);
towerOfHanoi(n-1, CENTER, RIGHT, LEFT);
}
}
int main()
{
int n;
printf("Enter no. of disks: ");
scanf("%d",&n);
printf("SOLUTION  (L=Left,R=Right,C=Ceter)\n");
towerOfHanoi(n,'L','R','C');
return 0;
}
```

Singly Linked List

```
// an example of singly linked list
#include<stdio.h>
#include<malloc.h>
struct Node
{
int num;
struct Node *next;
};
struct Node *start=NULL;
void addAtEnd(int num)
{
struct Node *t,*j;
t=(struct Node *)malloc(sizeof(struct Node));
t->num=num;
t->next=NULL;
if(start==NULL)
{
start=t;
}
else
{
j=start;
while(j->next!=NULL)
{
j=j->next;
}
j->next=t;
}
}
void insertAtTop(int num)
{
struct Node *t;
t=(struct Node *)malloc(sizeof(struct Node));
t->num=num;
t->next=NULL;
if(start==NULL)
{
start=t;
}
else
{
t->next=start;
start=t;
}
}
void insertAtPosition(int num,int pos)
{
struct Node *p1,*p2;
int x;
```

```
struct Node *t;
t=(struct Node *)malloc(sizeof(struct Node));
t->num=num;
t->next=NULL;
x=1;
p1=start;
while(x<pos && p1!=NULL)
{
p2=p1;
p1=p1->next;
x++;
}
if(p1==NULL)
{
if(start==NULL)
{
start=t;
}
else
{
p2->next=t;
}
}
else
{
if(p1==start)
{
t->next=start;
start=t;
}
else
{
t->next=p1;
p2->next=t;
}
}
}
void removeFromPosition(int pos)
{
struct Node *p1,*p2;
int x;
if(start==NULL)
{
printf("Invalid Position\n");
return;
}
x=1;
p1=start;
while(x<pos && p1!=NULL)
{
```

```
p2=p1;
p1=p1->next;
x++;
}
if(p1==NULL)
{
printf("Invalid position\n");
return;
}
if(p1==start)
{
start=start->next;
}
else
{
p2->next=p1->next;
}
free(p1);
}
void traverseTopToBottom()
{
struct Node *t;
t=start;
while(t!=NULL)
{
printf("%d\n",t->num);
t=t->next;
}
}
void traverseBottomToTop(struct Node *t)
{
if(t==NULL)
{
return;
}
traverseBottomToTop(t->next);
printf("%d\n",t->num);
}
int main()
{
int ch,num,pos;
while(1)
{
printf("1. Add a node at end\n");
printf("2. Insert a node at top\n");
printf("3. Insert a node at a position\n");
printf("4. Remove a node from a position\n");
printf("5. Traverse - Top To Bottom\n");
printf("6. Traverse - Bottom To Top\n");
printf("7. Exit\n");
```

```
printf("Enter your choice :");
scanf("%d",&ch);
if(ch==1)
{
printf("Enter a number to add at end : ");
scanf("%d",&num);
addAtEnd(num);
}
if(ch==2)
{
printf("Enter a number to insert at top : ");
scanf("%d",&num);
insertAtTop(num);
}
if(ch==3)
{
printf("Enter a number to insert : ");
scanf("%d",&num);
printf("Enter position : ");
scanf("%d",&pos);
insertAtPosition(num,pos);
}
if(ch==4)
{
printf("Enter position of the node to remove : ");
scanf("%d",&pos);
removeFromPosition(pos);
}
if(ch==5)
{
traverseTopToBottom();
}
if(ch==6)
{
traverseBottomToTop(start);
}
if(ch==7)
{
break;
}
}
return 0;
}
```

// Stack implementation using array (Technique 1)

```
#include<stdio.h>
int stack[10];
int top=0;
int upperBound=9;
int size=0;
void push(int num)
{
int i;
if(size==upperBound+1)
{
return; // stack full
}
i=size;
while(i>top)
{
stack[i]=stack[i-1];
i--;
}
stack[top]=num;
size++;
}
int pop()
{
int i,num;
if(size==0)
{
return 0; // stack empty
}
num=stack[top];
i=top;
while(i<size-1)
{
stack[i]=stack[i+1];
i++;
}
size--;
return num;
}
int isEmpty()
{
return size==0;
}
int isFull()
{
return size==upperBound+1;
}
int main()
{
int ch,num;
```

```
while(1)
{
printf("1. Push A Number On Stack\n");
printf("2. POP A Number From Stack\n");
printf("3. Exit\n");
printf("Enter your choice : ");
scanf("%d",&ch);
if(ch==1)
{
if(isFull())
{
printf("Error ! Stack Is Full\n");
}
else
{
printf("Enter Number To Push On Stack : ");
scanf("%d",&num);
if(num==0)
{
printf("Error ! Zero Cannot Be Pushed On Stack\n");
}
else
{
push(num);
printf("%d Pushed On Stack\n",num);
}
}
}
if(ch==2)
{
if(isEmpty())
{
printf("Error ! Stack Is Empty\n");
}
else
{
num=pop();
printf("%d Popped From Stack\n",num);
}
}
if(ch==3)
{
break;
}
}
return 0;
}
```

// Stack implementation using array (Technique 2)

```
#include<stdio.h>
int stack[10];
int lowerBound=0;
int upperBound=9;
int top;
void push(int num)
{
if(top==lowerBound)
{
return; // stack full
}
top--;
stack[top]=num;
}
int pop()
{
int num;
if(top==upperBound+1)
{
return 0; // stack empty
}
num=stack[top];
top++;
return num;
}
int isEmpty()
{
return top==upperBound+1;
}
int isFull()
{
return top==lowerBound;
}
int main()
{
int ch,num;
top=upperBound+1;
while(1)
{
printf("1. Push A Number On Stack\n");
printf("2. POP A Number From Stack\n");
printf("3. Exit\n");
printf("Enter your choice : ");
scanf("%d",&ch);
if(ch==1)
{
if(isFull())
{
printf("Error ! Stack Is Full\n");
```

```
}
else
{
printf("Enter Number To Push On Stack : ");
scanf("%d",&num);
if(num==0)
{
printf("Error ! Zero Cannot Be Pushed On Stack\n");
}
else
{
push(num);
printf("%d Pushed On Stack\n",num);
}
}
}
if(ch==2)
{
if(isEmpty())
{
printf("Error ! Stack Is Empty\n");
}
else
{
num=pop();
printf("%d Popped From Stack\n",num);
}
}
if(ch==3)
{
break;
}
}
return 0;
}
```

// implementation of stack using Linked List

```
#include<stdio.h>
#include<malloc.h>
struct Node
{
int num;
struct Node *next;
};
struct Node *top;
// we will use the word top in place of start
void push(int num)
{
struct Node *ptr;
```

```
ptr=(struct Node *)malloc(sizeof(struct Node));
ptr->num=num;
if(top==NULL)
{
ptr->next=NULL;
top=ptr;
}
else
{
ptr->next=top;
top=ptr;
}
}
int pop()
{
int num;
struct Node *ptr;
if(top==NULL)
{
return 0; // stack empty
}
num=top->num;
ptr=top;
top=top->next;
free(ptr);
return num;
}
int isEmpty()
{
return top==NULL;
}

int main()
{
int ch,num;
top=NULL;
while(1)
{
printf("1. Push A Number On Stack\n");
printf("2. POP A Number From Stack\n");
printf("3. Exit\n");
printf("Enter your choice : ");
scanf("%d",&ch);
if(ch==1)
{
printf("Enter Number To Push On Stack : ");
scanf("%d",&num);
if(num==0)
{
printf("Error ! Zero Cannot Be Pushed On Stack\n");
```

```
}
else
{
push(num);
printf("%d Pushed On Stack\n",num);
}
}

if(ch==2)
{
if(isEmpty())
{
printf("Error ! Stack Is Empty\n");
}
else
{
num=pop();
printf("%d Popped From Stack\n",num);
}
}
if(ch==3)
{
break;
}
}
return 0;
}
```

Application of stack

Converting infix expression to postfix expressions
Steps have been discussed in the classroom session

```
#include<stdio.h>
char stack[100];
int lowerBound=0;
int upperBound=99;
int top=100; // upper bound + 1
int size=0;
char postFix[101];
int postFixLowerBound=0;
void push(char op)
{
top--;
stack[top]=op;
size++;
}
char pop()
{
```

```
char op;
op=stack[top];
top++;
size--;
return op;
}
int isEmpty()
{
return top==upperBound+1;
}
int isOperator(char c)
{
return (c=='^' || c=='/' || c=='*' || c=='+' || c=='-');
}
int isOperand(char c)
{
if(isOperator(c)) return 0;
else return 1;
}
int getPrecedenceLevel(char op)
{
if(op=='^') return 3;
if(op=='/' || op=='*') return 2;
if(op=='+' || op=='-') return 1;
return 0; // this case won't arise but is neccessary to avoid warning from the compiler that a return
value is required
}
char getElementAtTop()
{
return stack[top]; // the element has not been popped
}
void appendToPostFixString(char c)
{
postFix[postFixLowerBound]=c;
postFixLowerBound++;
postFix[postFixLowerBound]='\0';
}
int main()
{
char infix[101];
char c,op;
int i;
printf("Enter an infix expression\n");
scanf("%s",infix);
i=0;
while(infix[i]!='\0')
{
c=infix[i];
i++;
if(c=='(')
```

```
{
push(c);
continue;
}
if(c==')')
{
while(1)
{
op=pop();
if(op=='(')
{
break;
}
appendToPostFixString(op);
}
continue;
}
if(isOperand(c))
{
appendToPostFixString(c);
continue;
}
if(isOperator(c))
{
if(isEmpty())
{
push(c);
}
else
{
op=getElementAtTop();
if(op=='(')
{
push(c);
}
else
{
if(getPrecedenceLevel(op)>=getPrecedenceLevel(c))
{
while(1)
{
if(isEmpty())
{
break;
}
op=getElementAtTop();
if(op=='(')
{
break;
}
```

```
if(getPrecedenceLevel(op)<getPrecedenceLevel(c))
{
break;
}
op=pop();
appendToPostFixString(op);
}
push(c);
}
else
{
push(c);
}
}
}
continue;
}
}// while ends
while(1)
{
if(isEmpty())
{
break;
}
op=pop();
appendToPostFixString(op);
}
printf("Post Fix Expression is\n");
printf("%s",postFix);
return 0;
}
```

Evaluating post fix expression
Steps have been discussed in the classroom session

```
#include<stdio.h>
int stack[100];
int lowerBound=0;
int upperBound=99;
int top=100; // upperBound+1
void push(int num)
{
if(top==lowerBound) return;
top--;
stack[top]=num;
}
int pop()
{
int num;
if(top==upperBound+1) return 0;
```

```
num=stack[top];
top++;
return num;
}
int isEmpty()
{
return top==upperBound+1;
}
int isFull()
{
return top=-lowerBound;
}
int convertToInt(char c)
{
if(c>=48 || c<=57) return c-48;
else return 0;
}
int performOperation(int operand1,int operand2,char oper)
{
int x,result;
if(oper=='^')
{
result=1;
x=1;
while(x<=operand2)
{
result=result*operand1;
x++;
}
return result;
}
if(oper=='/') return operand1/operand2;
if(oper=='*') return operand1*operand2;
if(oper=='+') return operand1+operand2;
if(oper=='-') return operand1-operand2;

return 0; // this case won't arise if the input is correct
}
int isOperator(char c)
{
return (c=='^' || c=='/' || c=='*' || c=='+' || c=='-');
}
int isOperand(char c)
{
return !isOperator(c);
}
int main()
{
char postfix[101];
int result,operand1,operand2,i;
```

```
char c;
printf("Enter postfix expression\n");
scanf("%s",postfix);
i=0;
while(postfix[i]!='\0')
{
c=postfix[i];
i++;
if(isOperand(c))
{
push(convertToInt(c));
continue;
}
if(isOperator(c))
{
operand2=pop();
operand1=pop();
result=performOperation(operand1,operand2,c);
push(result);
continue;
}
}
result=pop();
printf("Solution : %d\n",result);
return 0;
}
```

Converting infix expression to prefix expressions
Steps have been discussed in the classroom session

```
#include<stdio.h>
char operatorStack[100];
int operatorStackLowerBound=0;
int operatorStackUpperBound=99;
int topOperatorStack=100; // upper bound + 1
char operandStack[100];
int operandStackLowerBound=0;
int operandStackUpperBound=99;
int topOperandStack=100; // upper bound + 1
void push(char g,char stack[],int *top)
{
(*top)--;
stack[*top]=g;
}
char pop(char stack[],int *top)
{
char op;
op=stack[*top];
(*top)++;
```

```
return op;
}
int isEmpty(char stack[],int top,int upperBound)
{
return top==upperBound+1;
}
int isOperator(char c)
{
return (c=='^' || c=='/' || c=='*' || c=='+' || c=='-');
}
int isOperand(char c)
{
if(isOperator(c)) return 0;
else return 1;
}
int getPrecedenceLevel(char op)
{
if(op=='^') return 3;
if(op=='/' || op=='*') return 2;
if(op=='+' || op=='-') return 1;
return 0; // this case won't arise but is neccessary to avoid warning from the compiler that a return
value is required
}
char getElementAtTop(char stack[],int top)
{
return stack[top]; // the element has not been popped
}
int main()
{
char infix[81],prefix[81],i;
char a,b;
printf("Enter the infix string ");
gets(infix);
i=0;
while(infix[i+1]!='\0') i++;
while(i>=0)
{
a=infix[i];
i--;
if(a==')')
{
push(a,operatorStack,&topOperatorStack);
continue;
}
if(a=='(')
{
while(1)
{
b=pop(operatorStack,&topOperatorStack);
if(b==')') break;
```

```
push(b,operandStack,&topOperandStack);
}
continue;
}
if(isOperand(a))
{
push(a,operandStack,&topOperandStack);
continue;
}
// the remaing case is that the char is an operator
if(isEmpty(operatorStack,topOperatorStack,operatorStackUpperBound))
{
push(a,operatorStack,&topOperatorStack);
continue;
}
b=getElementAtTop(operatorStack,topOperatorStack);
if(b==')')
{
push(a,operatorStack,&topOperatorStack);
continue;
}
if(getPrecedenceLevel(b)>getPrecedenceLevel(a))
{
while(1)
{
b=pop(operatorStack,&topOperatorStack);
push(b,operandStack,&topOperandStack);
if(isEmpty(operatorStack,topOperatorStack,operatorStackUpperBound)) break;
b=getElementAtTop(operatorStack,topOperatorStack);
if(b==')') break;
if(getPrecedenceLevel(b)<=getPrecedenceLevel(a)) break;
}
push(a,operatorStack,&topOperatorStack);
}
else
{
push(a,operatorStack,&topOperatorStack);
}
}
while(1)
{
if(isEmpty(operatorStack,topOperatorStack,operatorStackUpperBound)) break;
a=pop(operatorStack,&topOperatorStack);
push(a,operandStack,&topOperandStack);
}
i=0;
while(1)
{
if(isEmpty(operandStack,topOperandStack,operandStackUpperBound)) break;
a=pop(operandStack,&topOperandStack);
```

```
prefix[i]=a;
i++;
}
prefix[i]='\0';
printf("Prefix expression\n%s\n",prefix);
return 0;
}
```

Evaluating prefix expression
Steps have been discussed in the classroom session

```
#include<stdio.h>
int stack[100];
int lowerBound=0;
int upperBound=99;
int top=100; // upperBound+1
void push(int num)
{
if(top==lowerBound) return;
top--;
stack[top]=num;
}
int pop()
{
int num;
if(top==upperBound+1) return 0;
num=stack[top];
top++;
return num;
}
int isEmpty()
{
return top==upperBound+1;
}
int isFull()
{
return top=-lowerBound;
}
int convertToInt(char c)
{
if(c>=48 || c<=57) return c-48;
else return 0;
}
int performOperation(int operand1,int operand2,char oper)
{
int x,result;
if(oper=='^')
{
result=1;
```

```
x=1;
while(x<=operand2)
{
result=result*operand1;
x++;
}
return result;
}
if(oper=='/') return operand1/operand2;
if(oper=='*') return operand1*operand2;
if(oper=='+') return operand1+operand2;
if(oper=='-') return operand1-operand2;

return 0; // this case won't arise if the input is correct
}
int isOperator(char c)
{
return (c=='^' || c=='/' || c=='*' || c=='+' || c=='-');
}
int isOperand(char c)
{
return !isOperator(c);
}
int main()
{
char prefix[101];
int result,operand1,operand2,i;
char c;
printf("Enter prefix expression\n");
scanf("%s",prefix);
i=0;
while(prefix[i+1]!='\0') i++;
while(i>=0)
{
c=prefix[i];
i--;
if(isOperand(c))
{
push(convertToInt(c));
continue;
}
if(isOperator(c))
{
operand1=pop();
operand2=pop();
result=performOperation(operand1,operand2,c);
push(result);
continue;
}
}
```

```
result=pop();
printf("Solution : %d\n",result);
return 0;
}
```

Queue
Array implementation of queues

```
#include<stdio.h>
int queue[10];
int lowerBound=0;
int upperBound=9;
int front=-1;
int rear=-1;
void addToQueue(int num)
{
if(rear==upperBound) return; // queue full
rear++;
queue[rear]=num;
if(front==-1) front=0;
}
int removeFromQueue()
{
int num,i;
if(rear==-1) return 0; // queue empty
num=queue[front];
i=0;
while(i<rear)
{
queue[i]=queue[i+1];
i++;
}
rear--;
if(rear==-1) front=-1;
return num;
}
int isQueueFull()
{
return rear==upperBound;
}
int isQueueEmpty()
{
return rear==-1;
}
int main()
{
int ch,num;
while(1)
{
printf("1 Add To Queue\n");
```

```
printf("2 Remove From Queue\n");
printf("3 Exit\n");
printf("Enter your choice ");
scanf("%d",&ch);
if(ch==1)
{
if(isQueueFull())
{
printf("Queue is full\n");
}
else
{
printf("Enter number to add to queue ");
scanf("%d",&num);
if(num==0)
{
printf("Cannot add zero to queue\n");
}
else
{
addToQueue(num);
printf("%d added to queue\n",num);
}
}
}
if(ch==2)
{
if(isQueueEmpty())
{
printf("Queue is empty\n");
}
else
{
num=removeFromQueue();
printf("%d removed from queue\n",num);
}
}
if(ch==3)
{
break;
}
}
return 0;
}
```

A linked implementation of queue

```
#include<stdio.h>
#include<malloc.h>
struct Node
```

```
{
int num;
struct Node *next;
};
struct Node *start=NULL;
void addToQueue(int num)
{
struct Node *t,*j;
t=(struct Node *)malloc(sizeof(struct Node));
t->num=num;
t->next=NULL;
if(start==NULL)
{
start=t;
}
else
{
j=start;
while(j->next!=NULL)
{
j=j->next;
}
j->next=t;
}
}
int removeFromQueue()
{
struct Node *t;
int num;
if(start==NULL) return 0; // queue is empty
num=start->num;
t=start;
start=start->next;
free(t);
return num;
}
int isQueueEmpty()
{
return start==NULL;
}
int main()
{
int ch,num;
while(1)
{
printf("1 Add To Queue\n");
printf("2 Remove From Queue\n");
printf("3 Exit\n");
printf("Enter your choice ");
scanf("%d",&ch);
```

```
if(ch==1)
{
printf("Enter number to add to queue ");
scanf("%d",&num);
addToQueue(num);
printf("%d added to queue\n",num);
}
if(ch==2)
{
if(isQueueEmpty())
{
printf("Queue is empty\n");
}
else
{
num=removeFromQueue();
printf("%d removed from queue\n",num);
}
}
if(ch==3)
{
break;
}
}
return 0;
}
```

Circular queues

```
#include<stdio.h>
int queue[10];
int lowerBound=0;
int upperBound=9;
int front=-1;
int rear=-1;
int size=0;
void addToQueue(int num)
{
if(size==(upperBound-lowerBound)+1) return; // queue full
if(rear<upperBound)
{
rear++;
}
else
{
rear=lowerBound;
}
queue[rear]=num;
size++;
```

```
if(front==-1) front=0;
}
int removeFromQueue()
{
int num;
if(size==0) return 0; // queue empty
num=queue[front];
if(front<upperBound)
{
front++;
}
else
{
front=lowerBound;
}
size--;
return num;
}
int isQueueFull()
{
return size==(upperBound-lowerBound)+1;
}
int isQueueEmpty()
{
return size==0;
}
int main()
{
int ch,num;
while(1)
{
printf("1 Add To Queue\n");
printf("2 Remove From Queue\n");
printf("3 Exit\n");
printf("Enter your choice ");
scanf("%d",&ch);
if(ch==1)
{
if(isQueueFull())
{
printf("Queue is full\n");
}
else
{
printf("Enter number to add to queue ");
scanf("%d",&num);
if(num==0)
{
printf("Cannot add zero to queue\n");
}
```

```
else
{
addToQueue(num);
printf("%d added to queue\n",num);
}
}
}
if(ch==2)
{
if(isQueueEmpty())
{
printf("Queue is empty\n");
}
else
{
num=removeFromQueue();
printf("%d removed from queue\n",num);
}
}
if(ch==3)
{
break;
}
}
return 0;
}
```

Double ended queues (d-queue)

```
#include<stdio.h>
#include<malloc.h>
struct Node
{
int num;
struct Node *next;
};
struct Node *start=NULL;
void addAtRear(int num)
{
struct Node *t,*j;
t=(struct Node *)malloc(sizeof(struct Node));
t->num=num;
t->next=NULL;
if(start==NULL)
{
start=t;
}
else
{
```

```
j=start;
while(j->next!=NULL)
{
j=j->next;
}
j->next=t;
}
}
int removeFromFront()
{
struct Node *t;
int num;
if(start==NULL) return 0; // queue is empty
num=start->num;
t=start;
start=start->next;
free(t);
return num;
}

void addAtFront(int num)
{
struct Node *t,*j;
t=(struct Node *)malloc(sizeof(struct Node));
t->num=num;
t->next=NULL;
if(start==NULL)
{
start=t;
}
else
{
t->next=start;
start=t;
}
}
int removeFromEnd()
{
struct Node *p1,*p2;
int num;
if(start==NULL) return 0; // queue is empty
if(start->next==NULL)
{
num=start->num;
start=NULL;
}
else
{
p1=start;
while(p1->next!=NULL)
```

```
{
p2=p1;
p1=p1->next;
}
num=p1->num;
p2->next=NULL;
}
free(p1);
return num;
}
int isQueueEmpty()
{
return start==NULL;
}

int main()
{
int ch,num;
while(1)
{
printf("1 Add At Front\n");
printf("2 Remove From Front\n");
printf("3 Add At End\n");
printf("4 Remove From End\n");
printf("5 Exit\n");
printf("Enter your choice ");

scanf("%d",&ch);
if(ch==1)
{
printf("Enter number to add at front ");
scanf("%d",&num);
addAtFront(num);
printf("%d added at front\n",num);
}
if(ch==2)
{
if(isQueueEmpty())
{
printf("Queue is empty\n");
}
else
{
num=removeFromFront();
printf("%d removed from front\n",num);
}
}

if(ch==3)
{
```

```
printf("Enter number to add at end ");
scanf("%d",&num);
addAtRear(num);
printf("%d added at end\n",num);
}
if(ch==4)
{
if(isQueueEmpty())
{
printf("Queue is empty\n");
}
else
{
num=removeFromEnd();
printf("%d removed from end\n",num);
}
}
if(ch==5)
{
break;
}
}
return 0;
}
```

Priority Queues

```
//Ascending Priority Queue using linked list
#include<stdio.h>
#include<malloc.h>
struct Node
{
int num,priorityNumber;
struct Node *next;
};
struct Node *start=NULL;
void addToQueue(int num,int priorityNumber)
{
struct Node *t,*p1,*p2;
t=(struct Node *)malloc(sizeof(struct Node));
t->num=num;
t->priorityNumber=priorityNumber;
t->next=NULL;
if(start==NULL)
{
start=t;
}
else
{
```

```
p1=start;
while(p1!=NULL)
{
// change the greater than condition to less than
// to make this example of descending priority queue
if(p1->priorityNumber>t->priorityNumber)
{
break;
}
p2=p1;
p1=p1->next;
}
if(p1==NULL)
{
p2->next=t;
return;
}
if(p1==start)
{
t->next=start;
start=t;
return;
}
t->next=p1;
p2->next=t;
}
}
struct Node * removeFromQueue()
{
struct Node *t;
if(start==NULL) return 0; // queue is empty
t=start;
start=start->next;
return t;
}
int isQueueEmpty()
{
return start==NULL;
}
void traverseQueue()
{
struct Node *t;
if(start==NULL) return;
t=start;
while(t!=NULL)
{
printf("Number %4d --- Priority Number %4d\n",t->num,t->priorityNumber);
t=t->next;
}
}
```

```
int main()
{
int ch,num,priorityNumber;
struct Node *t;
while(1)
{
printf("1 Add To Queue\n");
printf("2 Remove From Queue\n");
printf("3 Print Queue\n");
printf("4 Exit\n");
printf("Enter your choice ");
scanf("%d",&ch);
if(ch==1)
{
printf("Enter number to add to queue ");
scanf("%d",&num);
printf("Enter priority number ");
scanf("%d",&priorityNumber);
addToQueue(num,priorityNumber);
printf("%d added to queue with priority number %d\n",num,priorityNumber);
}
if(ch==2)
{
if(isQueueEmpty())
{
printf("Queue is empty\n");
}
else
{
t=removeFromQueue();
printf("%d with priority number %d removed from queue\n",t->num,t->priorityNumber);
}
}
if(ch==3)
{
traverseQueue();
}
if(ch==4)
{
break;
}
}
return 0;
}
```

Two-way Header List - Doubly linked list

```
// an example of doubly linked list
#include<stdio.h>
#include<malloc.h>
struct Node
{
int num;
struct Node *next,*prev;
};
struct Node *start=NULL;
struct Node *end=NULL;
void addAtEnd(int num)
{
struct Node *t;
t=(struct Node *)malloc(sizeof(struct Node));
t->num=num;
t->next=NULL;
t->prev=NULL;
if(start==NULL)
{
start=t;
end=t;
}
else
{
end->next=t;
t->prev=end;
end=t;
}
}
void insertAtTop(int num)
{
struct Node *t;
t=(struct Node *)malloc(sizeof(struct Node));
t->num=num;
t->next=NULL;
t->prev=NULL;
if(start==NULL)
{
start=t;
end=t;
}
else
{
t->next=start;
start->prev=t;
start=t;
}
}
void insertAtPosition(int num,int pos)
```

```
{
struct Node *p1;
int x;
struct Node *t;
t=(struct Node *)malloc(sizeof(struct Node));
t->num=num;
t->next=NULL;
t->prev=NULL;
x=1;
p1=start;
while(x<pos && p1!=NULL)
{
p1=p1->next;
x++;
}
if(p1==NULL)
{
if(start==NULL)
{
start=t;
end=t;
}
else
{
end->next=t;
t->prev=end;
end=t;
}
}
else
{
if(p1==start)
{
t->next=start;
start->prev=t;
start=t;
}
else
{
t->next=p1;
t->prev=p1->prev;
p1->prev->next=t;
p1->prev=t;
}
}
}
void removeFromPosition(int pos)
{
struct Node *p1;
int x;
```

```
if(start==NULL)
{
printf("Invalid Position\n");
return;
}
x=1;
p1=start;
while(x<pos && p1!=NULL)
{
p1=p1->next;
x++;
}
if(p1==NULL)
{
printf("Invalid position\n");
return;
}
if(p1==start && p1==end)
{
start=NULL;
end=NULL;
}
else
{
if(p1==start)
{
start=start->next;
start->prev=NULL;
}
else
{
if(p1==end)
{
end=end->prev;
end->next=NULL;
}
else
{
p1->prev->next=p1->next;
p1->next->prev=p1->prev;
}
}
}
free(p1);
}
void traverseTopToBottom()
{
struct Node *t;
t=start;
while(t!=NULL)
```

```
{
printf("%d\n",t->num);
t=t->next;
}
}
void traverseBottomToTop()
{
struct Node *t;
t=end;
while(t!=NULL)
{
printf("%d\n",t->num);
t=t->prev;
}
}
int main()
{
int ch,num,pos;
while(1)
{
printf("1. Add a node at end\n");
printf("2. Insert a node at top\n");
printf("3. Insert a node at a position\n");
printf("4. Remove a node from a position\n");
printf("5. Traverse - Top To Bottom\n");
printf("6. Traverse - Bottom To Top\n");
printf("7. Exit\n");
printf("Enter your choice :");
scanf("%d",&ch);
if(ch==1)
{
printf("Enter a number to add at end : ");
scanf("%d",&num);
addAtEnd(num);
}
if(ch==2)
{
printf("Enter a number to insert at top : ");
scanf("%d",&num);
insertAtTop(num);
}
if(ch==3)
{
printf("Enter a number to insert : ");
scanf("%d",&num);
printf("Enter position : ");
scanf("%d",&pos);
insertAtPosition(num,pos);
}
if(ch==4)
```

```
{
printf("Enter position of the node to remove : ");
scanf("%d",&pos);
removeFromPosition(pos);
}
if(ch==5)
{
traverseTopToBottom();
}
if(ch==6)
{
traverseBottomToTop();
}
if(ch==7)
{
break;
}
}
return 0;
}
```

Linked List in Array,

```
#include<stdio.h>
struct Node
{
int num;
int next;
char used;
};
int lowerBound=0;
int upperBound=4; // after testing for linked list full case, change 4 to 99
struct Node nodes[100];
int start=-1;
void initializer()
{
int i;
for(i=lowerBound;i<=upperBound;i++)
{
nodes[i].next=-1;
nodes[i].used='N';
}
start=-1;
}
int getIndexOfFreeNode()
{
int i;
for(i=lowerBound;i<=upperBound;i++)
{
```

```
if(nodes[i].used=='N') return i;
}
return -1;
}
void addAtEnd(int num)
{
int indexOfFreeNode,j;
indexOfFreeNode=getIndexOfFreeNode();
if(indexOfFreeNode==-1)
{
printf("Cannot add, linked list is full\n");
return;
}
nodes[indexOfFreeNode].num=num;
nodes[indexOfFreeNode].used='Y';
if(start==-1)
{
start=indexOfFreeNode;
}
else
{
j=start;
while(nodes[j].next!=-1)
{
j=nodes[j].next;
}
nodes[j].next=indexOfFreeNode;
}
}
void insertAtTop(int num)
{
int indexOfFreeNode,j;
indexOfFreeNode=getIndexOfFreeNode();
if(indexOfFreeNode==-1)
{
printf("Cannot add, linked list is full\n");
return;
}
nodes[indexOfFreeNode].num=num;
nodes[indexOfFreeNode].used='Y';
if(start==-1)
{
start=indexOfFreeNode;
}
else
{
nodes[indexOfFreeNode].next=start;
start=indexOfFreeNode;
}
}
```

```
void insertAtPosition(int num,int pos)
{
int x,p1,p2;
int indexOfFreeNode,j;
indexOfFreeNode=getIndexOfFreeNode();
if(indexOfFreeNode==-1)
{
printf("Cannot add, linked list is full\n");
return;
}
nodes[indexOfFreeNode].num=num;
nodes[indexOfFreeNode].used='Y';
if(start==-1)
{
start=indexOfFreeNode;
return;
}
if(pos<=0) pos=1;
p1=start;
x=1;
while(x<pos && p1!=-1)
{
p2=p1;
p1=nodes[p1].next;
x++;
}
if(p1==-1) // add at end
{
nodes[p2].next=indexOfFreeNode;
}
else
{
if(p1==start) // insert at top
{
nodes[indexOfFreeNode].next=start;
start=indexOfFreeNode;
}
else  // insert in between
{
nodes[indexOfFreeNode].next=p1;
nodes[p2].next=indexOfFreeNode;
}
}
}
void removeFromPosition(int pos)
{
int p1,p2,x;
if(pos<=0 || start==-1)
{
printf("Invalid position\n");
```

```
return;
}
p1=start;
x=1;
while(x<pos && p1!=-1)
{
p2=p1;
p1=nodes[p1].next;
x++;
}
if(p1==-1)
{
printf("Invalid position\n");
return;
}
if(p1==start)  // remove first
{
start=nodes[start].next;
}
else
{
nodes[p2].next=nodes[p1].next;
}
nodes[p1].used='N';
nodes[p1].next=-1;
}
void traverseTopToBottom()
{
int i;
i=start;
while(i!=-1)
{
printf("%d\n",nodes[i].num);
i=nodes[i].next;
}
}
void traverseBottomToTop(int k)
{
if(k==-1) return;
traverseBottomToTop(nodes[k].next);
printf("%d\n",nodes[k].num);
}
int main()
{
int ch,num,pos;
initializer();
while(1)
{
printf("1. Add a node at end\n");
printf("2. Insert a node at top\n");
```

```
printf("3. Insert a node at a position\n");
printf("4. Remove a node from a position\n");
printf("5. Traverse - Top To Bottom\n");
printf("6. Traverse - Bottom To Top\n");
printf("7. Exit\n");
printf("Enter your choice :");
scanf("%d",&ch);
if(ch==1)
{
printf("Enter a number to add at end : ");
scanf("%d",&num);
addAtEnd(num);
}
if(ch==2)
{
printf("Enter a number to insert at top : ");
scanf("%d",&num);
insertAtTop(num);
}
if(ch==3)
{
printf("Enter a number to insert : ");
scanf("%d",&num);
printf("Enter position : ");
scanf("%d",&pos);
insertAtPosition(num,pos);
}
if(ch==4)
{
printf("Enter position of the node to remove : ");
scanf("%d",&pos);
removeFromPosition(pos);
}
if(ch==5)
{
traverseTopToBottom();
}
if(ch==6)
{
traverseBottomToTop(start);
}
if(ch==7)
{
break;
}
}
return 0;
}
```

Application of singly linked list : Polynomial Arithmetic

```
#include<string.h>
#include<stdio.h>
#include<malloc.h>
#include<stdio.h>
#include<stdlib.h>
struct Node
{
char oper;
int coefficient;
char var;
int power;
struct Node *next;
};
struct Node *start1;
struct Node *start2;
struct Node *start3;
void parsePolynomial(char *p,struct Node **start)
{
struct Node *t,*j;
char tmp[21],oper,var;
char *q;
int i,coefficient,power;
var='0';
oper='+';
coefficient=1;
power=1;
while(*p!='\0')
{
if(*p=='+' || *p=='-')
{
q=p+1;
}
else
{
q=p;
}
while(*q!='\0' && (*q!='+' && *q!='-')) q++;
if(*p=='+' || *p=='-')
{
oper=*p;
p++;
}
if(*p>=48 && *p<=57)
{
i=0;
while(*p>=48 && *p<=57)
{
tmp[i]=*p;
i++;
p++;
```

```
}
tmp[i]='\0';
coefficient=atoi(tmp);
}
if(p==q)
{
var='0';
power=1;
}
else
{
var=*p;
p++;
if(p==q)
{
power=1;
}
else
{
if(*p=='^') p++;
i=0;
while(*p>=48 && *p<=57)
{
tmp[i]=*p;
i++;
p++;
}
tmp[i]='\0';
power=atoi(tmp);
}
}
t=(struct Node *)malloc(sizeof(struct Node));
t->oper=oper;
t->coefficient=coefficient;
t->var=var;
t->power=power;
t->next=NULL;
if(*start==NULL)
{
*start=t;
}
else
{
j=*start;
while(j->next!=NULL) j=j->next;
j->next=t;
}
var='0';
oper='+';
coefficient=1;
```

```
power=1;
}
}
void addPolynomials(struct Node **start1,struct Node **start2,struct Node **start3)
{
struct Node *t,*tmp,*p1,*p2,*t3,*e3;
int c1,c2,sum;
t=*start1;
while(t!=NULL)
{
p1=*start2;
while(p1!=NULL)
{
if(p1->var==t->var && t->power==p1->power) break;
p2=p1;
p1=p1->next;
}
if(p1==NULL) // not found
{
t3=(struct Node *)malloc(sizeof(struct Node));
t3->oper=t->oper;
t3->coefficient=t->coefficient;
t3->var=t->var;
t3->power=t->power;
t3->next=NULL;

if(*start3==NULL)
{
*start3=t3;
}
else
{
e3->next=t3;
}
e3=t3;
}
else
{
c1=t->coefficient;
if(t->oper=='-')
{
c1=c1*(-1);
}
c2=p1->coefficient;
if(p1->oper=='-')
{
c2=c2*(-1);
}
sum=c1+c2;
if(sum!=0)
```

```
{
t3=(struct Node *)malloc(sizeof(struct Node));
t3->oper=t->oper;
if(sum>=0)
{
t3->oper='+';
}
else
{
t3->oper='-';
sum=sum*(-1);
}
t3->coefficient=sum;
t3->var=t->var;
t3->power=t->power;
t3->next=NULL;
if(*start3==NULL)
{
*start3=t3;
}
else
{
e3->next=t3;
}
e3=t3;
}
if(p1==*start2)
{
*start2=(*start2)->next;
}
else
{
p2->next=p1->next;
}
free(p1);
}
tmp=t;
t=t->next;
free(tmp);
}
t=*start2;
while(t!=NULL)
{
t3=(struct Node *)malloc(sizeof(struct Node));
t3->var=t->var;
t3->power=t->power;
t3->oper=t->oper;
t3->coefficient=t->coefficient;
t3->next=NULL;
if(*start3==NULL)
```

```
{
*start3=t3;
}
else
{
e3->next=t3;
}
e3=t3;
tmp=t;
t=t->next;
free(tmp);
}
*start1=NULL;
*start2=NULL;
}
void convertToPolynomial(struct Node **start,char *polynomial)
{
char tmp[11];
int i;
struct Node *t;
i=0;
while((*start)!=NULL)
{
t=*start;
if(i==0)
{
if(t->oper=='-')
{
polynomial[i]=t->oper;
i++;
}
}
else
{
polynomial[i]=t->oper;
i++;
}
if(t->coefficient>1)
{
itoa(t->coefficient,tmp,10);
strcpy(polynomial+i,tmp);
i=i+strlen(tmp);
}
if(t->var!='0')
{
polynomial[i]=t->var;
i++;
}
if(t->power>1)
{
```

```
itoa(t->power,tmp,10);
polynomial[i]='^';
i++;
strcpy(polynomial+i,tmp);
i=i+strlen(tmp);
}
*start=(*start)->next;
free(t);
}
polynomial[i]='\0';
*start=NULL;
i=0;
}
int main()
{
char firstPolynomial[81],secondPolynomial[81],resultPolynomial[81];
start1=NULL;
start2=NULL;
start3=NULL;
printf("Enter first polynomial (one variable only) eg. x^2+3x^3-5 \n");
gets(firstPolynomial);
fflush(stdin);
parsePolynomial(firstPolynomial,&start1);
printf("Enter second polynomial (one variable only) eg. x^2+3x^3-5 \n");
gets(secondPolynomial);
fflush(stdin);
parsePolynomial(secondPolynomial,&start2);
addPolynomials(&start1,&start2,&start3);
convertToPolynomial(&start3,resultPolynomial);
if(strlen(resultPolynomial)>0)
{
printf("Sum is : %s\n",resultPolynomial);
}
else
{
printf("Sum is (nil)");
}
return 0;
}
```

Binary Search Tree (BST)

```
#include<stdio.h>
#include<malloc.h>
struct Node
{
int num;
struct Node *left,*right;
};

// the stack implementation can be converted to a singly linked list implementation if you don't want the limit of 1000 numbers.
```

```
struct Node  *stack[1000];
int upperBound=999;
int top=1000; // upperBound+1
void push(struct Node *t)
{
top--;
stack[top]=t;
}
struct Node * pop()
{
struct Node *t;
t=stack[top];
top++;
return t;
}
struct Node * getElementAtTop()
{
return stack[top];
}

int isEmpty()
{
return top==upperBound+1;
}

struct Node *start=NULL;
void addNode(int num)
{
struct Node *t,*j;
t=(struct Node *)malloc(sizeof(struct Node));
t->num=num;
t->left=NULL;
t->right=NULL;
if(start==NULL)
{
start=t;
}
else
{
j=start;
while(1)
{
if(t->num<j->num)
{
if(j->left==NULL)
{
j->left=t;
break;
}
else
```

```
{
j=j->left;
}
}
else
{
if(j->right==NULL)
{
j->right=t;
break;
}
else
{
j=j->right;
}
}
}
}
}
void removeNode(int num)
{
struct Node *t,*j,*e,*f;
t=start;
while(t!=NULL)
{
if(num==t->num)
{
break;
}
j=t;
if(num<t->num)
{
t=t->left;
}
else
{
t=t->right;
}
}
if(t==NULL)
{
printf("%d not found\n",num);
return;
}
if(t==start)
{
if(t->left==NULL && t->right==NULL)
{
start=NULL;
free(t);
```

```
return;
}
if(t->left==NULL)
{
start=t->right;
free(t);
return;
}
if(t->right==NULL)
{
start=t->left;
free(t);
return;
}
e=t->left;
f=t->right;
while(f->left!=NULL)
{
f=f->left;
}
f->left=e;
start=t->right;
free(t);
return;
}
if(t->num<j->num)
{
if(t->left==NULL && t->right==NULL)
{
j->left=NULL;
free(t);
return;
}
if(t->left==NULL)
{
j->left=t->right;
free(t);
return;
}
if(t->right==NULL)
{
j->left=t->left;
free(t);
return;
}
e=t->left;
f=t->right;
while(f->left!=NULL)
{
f=f->left;
```

```
}
f->left=e;
j->left=t->right;
free(t);
}
else
{
if(t->left==NULL && t->right==NULL)
{
j->right=NULL;
free(t);
return;
}
if(t->left==NULL)
{
j->right=t->right;
free(t);
return;
}
if(t->right==NULL)
{
j->right=t->left;
free(t);
return;
}
e=t->left;
f=t->right;
while(f->left!=NULL)
{
f=f->left;
}
f->left=e;
j->right=t->right;
free(t);
}
}
void inOrder(struct Node *t)
{
if(t==NULL)
{
return;
}
inOrder(t->left);
printf("%d\n",t->num);
inOrder(t->right);
}
void inOrderUsingStack()
{
struct Node *t;
t=start;
```

```
while(1)
{
if(t!=NULL)
{
push(t);
t=t->left;
}
else
{
if(isEmpty()) break;
t=pop();
printf("%d\n",t->num);
t=t->right;
}
}
}
void preOrder(struct Node *t)
{
if(t==NULL)
{
return;
}
printf("%d\n",t->num);
preOrder(t->left);
preOrder(t->right);
}

void preOrderUsingStack()
{
struct Node *t;
t=start;
while(1)
{
if(t!=NULL)
{
printf("%d\n",t->num);
push(t);
t=t->left;
}
else
{
if(isEmpty()) break;
t=pop();
t=t->right;
}
}
}
void postOrder(struct Node *t)
{
if(t==NULL)
```

```
{
return;
}
postOrder(t->left);
postOrder(t->right);
printf("%d\n",t->num);
}

void postOrderUsingStack()
{
struct Node *t;
if(start==NULL) return;
t=start;
while(1)
{
while (t!=NULL)
{
if (t->right!=NULL)
{
push(t->right);
}
push(t);
t=t->left;
}
t=pop(stack);
if(t->right!=NULL && getElementAtTop()==t->right)
{
pop();
push(t);
t=t->right;
}
else
{
printf("%d\n",t->num);
t=NULL;
}
if(isEmpty()) break;
}
}
int main()
{
int ch,num;
while(1)
{
printf("1. Add a number\n");
printf("2. Remove a number\n");
printf("3. Inorder Traversal (using recursion) \n");
printf("4. Preorder Traversal (using recursion)\n");
printf("5. Postorder Traversal (using recursion)\n");
printf("6. Inorder traversal (using Stack)\n");
```

```
printf("7. Preorder traversal (using Stack)\n");
printf("8. Postorder traversal (using Stack)\n");
printf("9. Exit\n");
printf("Enter your choice :");
scanf("%d",&ch);
if(ch==1)
{
printf("Enter number to add : ");
scanf("%d",&num);
addNode(num);
}
if(ch==2)
{
printf("Enter number to remove : ");
scanf("%d",&num);
removeNode(num);
}
if(ch==3)
{
inOrder(start);
}
if(ch==4)
{
preOrder(start);
}
if(ch==5)
{
postOrder(start);
}
if(ch==6)
{
inOrderUsingStack();
}
if(ch==7)
{
preOrderUsingStack();
}
if(ch==8)
{
postOrderUsingStack();
}
if(ch==9)
{
break;
}
}
return 0;
}
```

Postfix to Expression tree

```
#include<stdio.h>
#include<malloc.h>
struct Node
{
char element;
struct Node *left,*right;
};
struct Node *stack[1000];
int lowerBound=0;
int upperBound=999;
int top=1000;
int isEmpty()
{
return top==upperBound+1;
}
void push(struct Node *t)
{
top--;
stack[top]=t;
}
struct Node * pop()
{
struct Node *t;
t=stack[top];
top++;
return t;
}
int isOperator(char c)
{
return (c=='/' || c=='*' || c=='+' || c=='-');
}
int isOperand(char c)
{
if(isOperator(c)) return 0;
else return 1;
}
int getPrecedenceLevel(char op)
{
if(op=='/' || op=='*') return 2;
if(op=='+' || op=='-') return 1;
return 0;
}
struct Node *start=NULL;
void createExpressionTree(char *postFix)
{
struct Node *t,*e1,*e2;
int i;
char c;
i=0;
while(postFix[i]!='\0')
```

```
{
c=postFix[i];
if(isOperand(c))
{
t=(struct Node *)malloc(sizeof(struct Node));
t->element=c;
t->left=NULL;
t->right=NULL;
push(t);
}
else // (c contains an operator)
{
t=(struct Node *)malloc(sizeof(struct Node));
t->element=c;
e1=pop();
e2=pop();
t->left=e2;
t->right=e1;
push(t);
}
i++;
}
start=pop();
}
void inOrder(struct Node *t)
{
if(t==NULL) return;
if(t->left!=NULL && t->right!=NULL) printf("(");
inOrder(t->left);
printf("%c",t->element);
inOrder(t->right);
if(t->left!=NULL && t->right!=NULL) printf(")");
}
int main()
{
char postFix[81];
printf("Enter a postfix expression");
gets(postFix);
fflush(stdin);
createExpressionTree(postFix);
inOrder(start);
return 0;
}
```

Level order traversal in a Binary Tree

```
#include<stdio.h>
#include<malloc.h>
struct Node
{
int num;
struct Node *left,*right;
```

```
};
// Following implementation of queue can be converted to singly linked list
// if you don't want to fix the size of the queue
struct Node *queue[1000];
int lowerBound=0;
int upperBound=9;
int front=-1;
int rear=-1;
void addToQueue(struct Node *t) // enqueue
{
if(rear==upperBound) return; // queue full
rear++;
queue[rear]=t;
if(front==-1) front=0;
}
struct Node * removeFromQueue() // dequeue
{
struct Node *t;
if(rear==-1) return NULL; // queue empty
t=queue[front];
front++;
if(front>rear)
{
front=-1;
rear=-1;
}
return t;
}
struct Node *start=NULL;
void addNode(int num)
{
struct Node *t,*j;
t=(struct Node *)malloc(sizeof(struct Node));
t->num=num;
t->left=NULL;
t->right=NULL;
if(start==NULL)
{
start=t;
}
else
{
j=start;
while(1)
{
if(t->num<j->num)
{
if(j->left==NULL)
{
j->left=t;
```

```
break;
}
else
{
j=j->left;
}
}
else
{
if(j->right==NULL)
{
j->right=t;
break;
}
else
{
j=j->right;
}
}
}
}
}
int getHeightOfTheTree(struct Node *t)
{
int leftHeight,rightHeight;
if(t==NULL) return 0;
leftHeight=getHeightOfTheTree(t->left);
rightHeight=getHeightOfTheTree(t->right);
if(leftHeight>rightHeight) return leftHeight+1;
else return rightHeight+1;
}

void traverseLevel(struct Node* t,int level)
{
if(t==NULL)return;
if(level==1)printf("%d\n", t->num);
else
{
if(level>1)
{
traverseLevel(t->left, level-1);
traverseLevel(t->right, level-1);
}
}
}
void levelOrder(struct Node *s)
{
int i;
int height=getHeightOfTheTree(s);
for(i=1;i<=height;i++)
```

```
traverseLevel(s,i);
}
 void levelOrderUsingQueue(struct Node *s)
{
struct Node *t;
t=start;
while(t!=NULL)
{
printf("%d\n",t->num);
if(t->left!=NULL)addToQueue(t->left);
if(t->right!=NULL)addToQueue(t->right);
t=removeFromQueue();
}
}
int main()
{
int ch,num,height;
while(1)
{
printf("1. Add a number\n");
printf("2 Print the height of the tree\n");
printf("3. Level order Traversal (Using Recursion) \n");
printf("4. Level order Traversal (Using Queue) \n");
printf("5. Exit\n");
printf("Enter your choice :");
scanf("%d",&ch);
if(ch==1)
{
printf("Enter number to add : ");
scanf("%d",&num);
addNode(num);
}
if(ch==2)
{
height=getHeightOfTheTree(start);
printf("Height of the tree is %d\n",height);
}
if(ch==3) { levelOrder(start); }
if(ch==4) { levelOrderUsingQueue(start); }
if(ch==5) { break; }
}
return 0;
}
```

Complete Binary Tree

```
// Determine  Complete binary tree
/*
Various theories for complete binary tree exists and it is not possible to
justify that which is the correct one.
As per classroom discussion we will stick to the following theory
number of nodes=(2^(h+1)-1)  ----- if tree with one node is of height 0
number of nodes=(2^h - 1) ------ if tree with one node is of height 1

(h+1) because we are considering that the height of the tree with only one node is zero.
In some examples in some books the same has been said as 1. (discussed in classroom session)
*/
#include<stdio.h>
#include<malloc.h>
struct Node
{
int num;
struct Node *left,*right;
};
struct Node *start=NULL;
void addNode(int num)
{
struct Node *t,*j;
t=(struct Node *)malloc(sizeof(struct Node));
t->num=num;
t->left=NULL;
t->right=NULL;
if(start==NULL)
{
start=t;
}
else
{
j=start;
while(1)
{
if(t->num<j->num)
{
if(j->left==NULL)
{
j->left=t;
break;
}
else
{
j=j->left;
}
}
else
{
if(j->right==NULL)
```

```
{
j->right=t;
break;
}
else
{
j=j->right;
}
}
}
}
}
int getHeightOfTheTree(struct Node *t)
{
int leftHeight,rightHeight;
if(t==NULL) return 0;
leftHeight=getHeightOfTheTree(t->left);
rightHeight=getHeightOfTheTree(t->right);
if(leftHeight>rightHeight) return leftHeight+1;
else return rightHeight+1;
}
int countNumberOfNodes(struct Node *t)
{
if(t==NULL) return 0;
return 1+countNumberOfNodes(t->left)+countNumberOfNodes(t->right);
}
int power(int n,int p)
{
int x,pow;
pow=1;
x=1;
while(x<=p)
{
pow=pow*n;
x++;
}
return pow;
}
int isCompleteBinaryTree(struct Node *s)
{
int height,nodeCount,pow;
height=getHeightOfTheTree(s);
nodeCount=countNumberOfNodes(s);
pow=power(2,height);
return nodeCount==pow-1;
}
int main()
{
int ch,num,nodeCount;
while(1)
```

```
{
printf("1. Add a number\n");
printf("2. Count number of nodes\n");
printf("3 Is complete binary tree\n");
printf("4. Exit\n");
printf("Enter your choice :");
scanf("%d",&ch);
if(ch==1)
{
printf("Enter number to add : ");
scanf("%d",&num);
addNode(num);
}
if(ch==2)
{
nodeCount=countNumberOfNodes(start);
printf("Number of nodes %d\n",nodeCount);
}
if(ch==3)
{
if(isCompleteBinaryTree(start)) printf("Complete binary tree\n");
else printf("Not a complete binary tree\n");
}
if(ch==4)
{
break;
}
}
return 0;
}
```

Array representation of Binary Search Tree

```
#include<stdio.h>
// -1 stands for NULL in my logic
int bst[1000];
int start=0;
void nullifyAll()
{
int i;
for(i=0;i<1000;i++) bst[i]=-1;
}
int getLeftIndex(int i)
{
return (i*2)+1;
}
int getRightIndex(int i)
{
return (i+1)*2;
}
void addNode(int num)
{
```

```
int j,leftIndex,rightIndex;
if(bst[start]==-1)
{
bst[start]=num;
return;
}
j=start;
while(1)
{
if(num<bst[j])
{
leftIndex=getLeftIndex(j);
if(bst[leftIndex]==-1)
{
bst[leftIndex]=num;
break;
}
else
{
j=leftIndex;
}
}
else
{
rightIndex=getRightIndex(j);
if(bst[rightIndex]==-1)
{
bst[rightIndex]=num;
break;
}
else
{
j=rightIndex;
}
}
}
}
void removeNode(int num)
{
int t,j,e,f,leftIndex,rightIndex;
t=start;
while(bst[t]!=-1)
{
if(num==bst[t])
{
break;
}
j=t;
if(num<bst[t])
{
```

```
t=getLeftIndex(t);
}
else
{
t=getRightIndex(t);
}
}
if(bst[t]==-1)
{
printf("%d Not found\n",num);
return;
}
// complete it as an assignment ( discussed in the classroom session )
}
void inOrder(int t)
{
if(bst[t]==-1) return;
inOrder(getLeftIndex(t));
printf("%d\n",bst[t]);
inOrder(getRightIndex(t));
}
void preOrder(int t)
{
if(bst[t]==-1) return;
printf("%d\n",bst[t]);
inOrder(getLeftIndex(t));
inOrder(getRightIndex(t));
}
void postOrder(int t)
{
if(bst[t]==-1) return;
inOrder(getLeftIndex(t));
inOrder(getRightIndex(t));
printf("%d\n",bst[t]);
}
int main()
{
int ch,num;
nullifyAll();
while(1)
{
printf("Binary search tree using array\n");
printf("1. Add a number\n");
printf("2. Remove a number\n");
printf("3. Inorder traversal\n");
printf("4. Preorder traversal\n");
printf("5. Postorder traversal\n");
printf("6. Exit\n");
printf("Enter your choice ");
scanf("%d",&ch);
```

```
if(ch==1)
{
printf("Enter a number to add to tree");
scanf("%d",&num);
if(num==-1)
{
printf("Number -1 cannot be added to binary tree\n");
}
else
{
addNode(num);
}
}
if(ch==2)
{
printf("Enter the number to remove from tree");
scanf("%d",&num);
if(num==-1)
{
printf("-1 not in tree\n");
}
else
{
removeNode(num);
}
}
if(ch==3)
{
inOrder(start);
}
if(ch==4)
{
preOrder(start);
}
if(ch==5)
{
postOrder(start);
}
if(ch==6)
{
break;
}
}
}
```

Threaded Binary Tree

```
#include<stdio.h>
#include<malloc.h>
struct Node
{
int num;
struct Node *left,*right;
int isLeftAThread,isRightAThread;
};
struct Node *start=NULL;
void addNode(int num)
{
struct Node *t,*p1,*p2;
t=(struct Node *)malloc(sizeof(struct Node));
t->num=num;
t->left=NULL;
t->right=NULL;
t->isLeftAThread=0; // 0 for false
t->isRightAThread=0; // 0 for false
if(start==NULL)
{
start=t;
}
else
{
p1=start;
while(p1!=NULL)
{
p2=p1;
if(t->num<p1->num)
{
if(p1->isLeftAThread)
{
break;
}
else
{
p1=p1->left;
}
}
else
{
if(p1->isRightAThread)
{
break;
}
else
{
p1=p1->right;
}
}
```

```
}

if(p1==NULL)
{
if(t->num<p2->num)
{
p2->left=t;
t->right=p2;
t->isRightAThread=1; // 1 for true
}
else
{
p2->right=t;
t->left=p2;
t->isLeftAThread=1; // 1 for true
}
}
else
{
if(t->num<p1->num)
{
t->left=p1->left;
t->isLeftAThread=1; // 1 for true
t->right=p1;
t->isRightAThread=1; // 1 for true
p1->left=t;
p1->isLeftAThread=0; // 0 for false
}
else
{
t->right=p1->right;
t->isRightAThread=1; // 1 for true
t->left=p1;
t->isLeftAThread=1; // 1 for
p1->right=t;
p1->isRightAThread=0; // 0 for false
}
}
}
}

void traverseTheTree()
{
struct Node *t,*j;
if(start==NULL)
{
printf("No nodes\n");
return;
}
t=start;
```

```
while(t!=NULL)
{
while(t!=NULL)
{
j=t;
if(t->isLeftAThread)
{
break;
}
else
{
t=t->left;
}
}
printf("%d right thread \n",j->num);
t=j->right;
while(j->isRightAThread)
{
printf("%d left thread\n",t->num);
j=t;
t=j->right;
}
}
}
int main()
{
int ch,num;
while(1)
{
printf("1. Add a number\n");
printf("2. Traverse the tree\n");
printf("3. Exit\n");
printf("Enter your choice :");
scanf("%d",&ch);
if(ch==1)
{
printf("Enter number to add : ");
scanf("%d",&num);
addNode(num);
}
if(ch==2) { traverseTheTree(); }
if(ch==3) { break; }
}
return 0;
}
```

Quick Sort

```
#include<stdio.h>
int partition(int *x,int lb,int ub)
{
int e,f,g,num;
num=x[lb];
e=lb;
f=ub;
while(1)
{
while(1)
{
if(e==ub || x[e]>num)
{
break;
}
e++;
}
while(1)
{
if(f==lb || x[f]<=num)
{
break;
}
f--;
}
if(e<f)
{
g=x[e];
x[e]=x[f];
x[f]=g;
}
else
{
g=x[lb];
x[lb]=x[f];
x[f]=g;
break;
}
}
return f;
}
void quickSort(int *x,int lb,int ub)
{
int partitionPoint;
if(ub<=lb)
{
return;
}
partitionPoint=partition(x,lb,ub);
```

```
quickSort(x,lb,partitionPoint-1);
quickSort(x,partitionPoint+1,ub);
}
int main()
{
int x[10],y;
for(y=0;y<=9;y++)
{
printf("Enter a number : ");
scanf("%d",&x[y]);
}
quickSort(x,0,9);
for(y=0;y<=9;y++)
{
printf("%d\n",x[y]);
}
return 0;
}
```

//Bubble Sort

```
#include<stdio.h>
void bubbleSort(int *x,int size)
{
int e,f,m,g;
m=size-2;
while(m>=0)
{
e=0;
f=1;
while(e<=m)
{
if(x[f]<x[e])
{
g=x[e];
x[e]=x[f];
x[f]=g;
}
e++;
f++;
}
m--;
}
}
int main()
{
int x[10],y;
for(y=0;y<=9;y++)
{
printf("Enter a number : ");
scanf("%d",&x[y]);
}
```

```
bubbleSort(x,10);
for(y=0;y<=9;y++)
{
printf("%d\n",x[y]);
}
return 0;
}
```

//Linear sort

```
#include<stdio.h>
void linearSort(int *x,int size)
{
int e,f,g;
e=0;
while(e<=size-2)
{
f=e+1;
while(f<=size-1)
{
if(x[f]<x[e])
{
g=x[e];
x[e]=x[f];
x[f]=g;
}
f++;
}
e++;
}
}
int main()
{
int x[10],y;
for(y=0;y<=9;y++)
{
printf("Enter a number : ");
scanf("%d",&x[y]);
}
linearSort(x,10);
for(y=0;y<=9;y++)
{
printf("%d\n",x[y]);
}
return 0;
}
```

// Selection Sort

```
#include<stdio.h>
void selectionSort(int *x,int size)
{
int e,f,m,g;
e=0;
```

```
while(e<=size-2)
{
m=e;
f=e+1;
while(f<=size-1)
{
if(x[f]<x[m])
{
m=f;
}
f++;
}
g=x[e];
x[e]=x[m];
x[m]=g;
e++;
}
}
int main()
{
int x[10],y;
for(y=0;y<=9;y++)
{
printf("Enter a number : ");
scanf("%d",&x[y]);
}
selectionSort(x,10);
for(y=0;y<=9;y++)
{
printf("%d\n",x[y]);
}
return 0;
}
```

// Radix Sort

```
#include<Stdio.h>
int queue[10][100];
int front[10]={-1,-1,-1,-1,-1,-1,-1,-1,-1,-1};
int rear[10]={-1,-1,-1,-1,-1,-1,-1,-1,-1,-1};
void addToQueue(int i,int num)
{
rear[i]++;
queue[i][rear[i]]=num;
if(front[i]==-1) front[i]=0;
}
int removeFromQueue(int i)
{
int num;
if(front[i]==-1) return -1; // queue is empty
num=queue[i][front[i]];
front[i]++;
```

```
if(front[i]>rear[i])
{
front[i]=-1;
rear[i]=-1;
}
return num;
}
int isEmpty(int i)
{
return front[i]==-1;
}
void radixSort(int x[],int lb,int ub)
{
int digitPlace,num,y,e,f,digit,z;
digitPlace=1;
e=10;
f=1;
while(digitPlace<=4)
{
y=lb;
while(y<=ub)
{
num=x[y];
digit=(num%e)/f;
addToQueue(digit,num);
y++;
}
y=0;
z=0;
while(y<=9)
{
while(1)
{
if(isEmpty(y))
{
break;
}
num=removeFromQueue(y);
x[z]=num;
z++;
}
y++;
}
e=e*10;
f=f*10;
digitPlace++;
}
}
int main()
{
```

```
int x[10],y,e,f;
for(y=0;y<=9;y++)
{
printf("Enter a number : ");
scanf("%d",&x[y]);
}
radixSort(x,0,9);
for(y=0;y<=9;y++)
{
printf("%d\n",x[y]);
}
return 0;
}
```

// Insertion sort

```
#include<stdio.h>
void insertionSort(int *x,int lb,int ub)
{
int p,j,num,y;
y=lb+1;
while(y<=ub)
{
p=y;
num=x[p];
j=p-1;
while(j>=lb)
{
if(num>=x[j])
{
break; // we have found the position for num
}
x[j+1]=x[j]; // shifting
j--;
p--;
}
x[p]=num;
y++;
}
}
int main()
{
int x[10],y;
for(y=0;y<=9;y++)
{
printf("Enter a number : ");
scanf("%d",&x[y]);
}
insertionSort(x,0,9);
for(y=0;y<=9;y++)
{
printf("%d\n",x[y]);
```

```
}
return 0;
}
```

// Merge sort

```
#include<stdio.h>
void merge(int a[], int low, int high, int mid)
{
int i, j, k, c[50];
i=low;
j=mid+1;
k=low;
while((i<=mid)&&(j<=high))
{
if(a[i]<a[j])
{
c[k]=a[i];
k++;
i++;
}
else
{
c[k]=a[j];
k++;
j++;
}
}
while(i<=mid)
{
c[k]=a[i];
k++;
i++;
}
while(j<=high)
{
c[k]=a[j];
k++;
j++;
}
for(i=low;i<k;i++)
{
a[i]=c[i];
}
}
void mergesort(int a[], int low, int high)
{
int mid;
if(low<high)
{
mid=(low+high)/2;
mergesort(a,low,mid);
```

```
mergesort(a,mid+1,high);
merge(a,low,high,mid);
}
}
int main()
{
int x[10],y;
for(y=0;y<=9;y++)
{
printf("Enter a number : ");
scanf("%d",&x[y]);
}
mergesort(x,0,9);
for(y=0;y<=9;y++) printf("%d\n",x[y]);
return 0;
}
```

// Heap sort

```
#include<stdio.h>
void heapSort(int *x,int size)
{
int k,j,i,m,g;
i=1;
while(i<size)
{
m=i;
do
{
k=(m-1)/2;
if(x[k]<x[m])
{
g=x[k];
x[k]=x[m];
x[m]=g;
}
m=k;
}while(m!=0);
i++;
}
for(j=size-1;j>=0;j--)
{
g=x[0];
x[0]=x[j];
x[j]=g;
k=0;
do{
m=2*k+1;
if((x[m]<x[m+1]) && m<j-1)
{
m++;
}
```

```
if(x[k]<x[m] && m<j)
{
g=x[k];
x[k]=x[m];
x[m]=g;
}
k=m;
}while(m<j);
}
}
int main()
{
int x[10],y;
for(y=0;y<=9;y++
{
printf("Enter a number");
scanf("%d",&x[y]);
}
heapSort(x,10);
for(y=0;y<=9;y++) printf("%d\n",x[y]);
return 0;
}
```

// Shell sort

```
#include<stdio.h>
void shellSort(int *x,int lb,int ub)
{
int p,j,num,y;
int size,distanceFactor;
size=ub-lb+1;
distanceFactor=size/2;
while(distanceFactor>0)
{
y=lb+distanceFactor;
while(y<=ub)
{
p=y;
num=x[p];
j=p-distanceFactor;
while(j>=lb)
{
if(num>=x[j])
{
break; // we have found the position for num
}
x[j+distanceFactor]=x[j]; // shifting
j=j-distanceFactor;
p=p-distanceFactor;
}
x[p]=num;
```

```
y=y+distanceFactor;
}
distanceFactor=distanceFactor/2;
}
}
int main()
{
int x[10],y;
for(y=0;y<=9;y++)
{
printf("Enter a number : ");
scanf("%d",&x[y]);
}
shellSort(x,0,9);
for(y=0;y<=9;y++)
{
printf("%d\n",x[y]);
}
return 0;
}
```

// **Sequential Search**

```
#include<stdio.h>
int sequentialSearch(int x[],int lowerBound,int upperBound,int lookFor)
{
int y;
y=lowerBound;
while(y<=upperBound)
{
if(x[y]==lookFor) return y;
y++;
}
return -1;
}
int main()
{
int x[10],y,lookFor,foundAt;
for(y=0;y<=9;y++)
{
printf("Enter a number");
scanf("%d",&x[y]);
}
printf("Search what : ");
scanf("%d",&lookFor);
foundAt=sequentialSearch(x,0,9,lookFor);
if(foundAt==-1) printf("%d not found",lookFor);
else printf("%d found at index %d",lookFor,foundAt);
}
```

```
// An example of Hash / linear probing - in case of collision
#include<stdio.h>
struct Student
{
int rollNumber;
char name[21];
};
// The application is based on the requirement analysis result that
// number of students will never exceed 1000
struct Student records[1000];

void clearGarbageValues()
{
int x;
for(x=0;x<=999;x++)
{
records[x].rollNumber=0;
records[x].name[0]='\0';
}
}
int recordCount=0;
int hashFunction(int data)
{
int key=data%1000; // 1000, because size of array is 1000 ( key index should be in between(0-
999));
// The hash function can be defined in many ways.
return key;
}
/* Collision Resolution Strategies (Discussed in classroom session)
 a) Linear probing (increment key with 1) till empty slot is found
 b) Quadratic probing
    (some mathematical formula till empty slot is found) (HF(data)+key^2)%arraySize
 c) Chaining ( a linked list against the found key slot to accomodate items with same keys )
*/
int getNewKeyAfterLinearProbing(int key)
{
key++;
if(key==1000) key=0;
while(records[key].rollNumber!=0)
{
key++;
if(key==1000) key=0;
}
return key;
}
int exists(int rollNumber)
{
int key,i;
key=hashFunction(rollNumber);
if(records[key].rollNumber!=rollNumber)
```

```
{
i=key+1;
if(i==1000) i=0;
while(i!=key)
{
if(records[i].rollNumber==rollNumber)
{
break;
}
i++;
if(i==1000) i=0;
}
if(i==key)
{
return -1;
}
else
{
key=i;
}
}
return key;
}
void searchStudent()
{
int rollNumber,key;
printf("Enter roll number");
scanf("%d",&rollNumber);
fflush(stdin);
if(rollNumber<=0)
{
printf("Invalid roll number\n");
return;
}
key=exists(rollNumber);
if(key==-1)
{
printf("Invalid roll number\n");
return;
}
printf("Name %s\n",records[key].name);
}
void addStudent()
{
int rollNumber,key;
printf("Enter roll Number");
scanf("%d",&rollNumber);
fflush(stdin);
if(rollNumber<=0)
{
```

```
printf("Invalid roll number\n");
return;
}
key=exists(rollNumber);
if(key!=-1)
{
printf("That roll number alloted to %s\n",records[key].name);
return;
}
key=hashFunction(rollNumber);
if(records[key].rollNumber!=0) // Slot is occupied
{
key=getNewKeyAfterLinearProbing(key);
}
records[key].rollNumber=rollNumber;
printf("Enter name : ");
gets(records[key].name);
fflush(stdin);
recordCount++;
}
int main()
{
int ch;
while(1)
{
printf("1. Add Student\n");
printf("2. Search Student\n");
printf("3. Exit\n");
printf("Enter your choice ");
scanf("%d",&ch);
fflush(stdin);
if((ch==1) && recordCount==1000)
{
printf("Cannot add more than 1000 records\n");
continue;
}
if(ch==2 && recordCount==0)
{
printf("Records not added\n");
continue;
}
if(ch==1) addStudent();
if(ch==2) searchStudent();
if(ch==3) break;
}
return 0;
}
```

// An example of Hash / Chaining - in case of collision

```
#include<stdio.h>
#include<malloc.h>
```

```
struct Student
{
int rollNumber;
char name[21];
struct Student *next;
};
struct Student *start[1000];

void clearGarbageValues()
{
int x;
for(x=0;x<=999;x++)
{
start[x]=NULL;
}
}
int hashFunction(int data)
{
int key=data%1000; // 1000, because size of array is 1000 ( key index should be in betwee(0-999));
// The hash function can be defined in many ways.
return key;
}

struct Student * exists(int rollNumber)
{
struct Student *t;
int key;
key=hashFunction(rollNumber);
t=start[key];
while(t!=NULL)
{
if(t->rollNumber==rollNumber) break;
t=t->next;
}
return t;
}
void searchStudent()
{
int rollNumber;
struct Student *t;
printf("Enter roll number");
scanf("%d",&rollNumber);
fflush(stdin);
if(rollNumber<=0)
{
printf("Invalid roll number\n");
return;
}
t=exists(rollNumber);
if(t==NULL)
```

```
{
printf("Invalid roll number\n");
return;
}
printf("Name %s\n",t->name);
}
void displayStudents()
{
int x;
struct Student *t;
x=0;
while(x<=999)
{
t=start[x];
while(t!=NULL)
{
printf("Roll number %d   Name %s\n",t->rollNumber,t->name);
t=t->next;
}
x++;
}
}
void addStudent()
{
int rollNumber,key;
struct Student *t,*j;
printf("Enter roll Number");
scanf("%d",&rollNumber);
fflush(stdin);
if(rollNumber<=0)
{
printf("Invalid roll number\n");
return;
}
t=exists(rollNumber);
if(t!=NULL)
{
printf("That roll number alloted to %s\n",t->name);
return;
}
key=hashFunction(rollNumber);
t=(struct Student *)malloc(sizeof(struct Student));
t->rollNumber=rollNumber;
t->next=NULL;
printf("Enter name : ");
gets(t->name);
fflush(stdin);
if(start[key]==NULL)
{
start[key]=t;
```

```
}
else
{
for(j=start[key];j->next!=NULL;j=j->next);
j->next=t;
}
}
int main()
{
int ch;
while(1)
{
printf("1. Add Student\n");   printf("2. Search Student\n");
printf("3. Display list\n"); // you can try to implement this option in previous example
printf("4. Exit\n");
printf("Enter your choice ");  scanf("%d",&ch); fflush(stdin);
if(ch==1) addStudent();
if(ch==2) searchStudent();
if(ch==3) displayStudents();
if(ch==4) break;
}
return 0;
}
// An example of maintaining indexes
#include<malloc.h>
#include<stdio.h>
struct Student
{
int rollNumber;
char name[21];
int age;
struct Student *next,*prev;
};
struct IndexNode
{
struct Student *student;
struct IndexNode *next,*prev;
};
struct Student *start,*end;
struct IndexNode *startOfNameIndex,*endOfNameIndex;
struct Student * getNewNode()
{
struct Student *t;
t=(struct Student *)malloc(sizeof(struct Student));
t->next=NULL;
t->prev=NULL;
return t;
}

struct IndexNode * getNewIndexNode()
```

```
{
struct IndexNode *ti;
ti=(struct IndexNode *)malloc(sizeof(struct IndexNode));
ti->next=NULL;
ti->prev=NULL;
return ti;
}

void addStudentNode(struct Student *t)
{
struct Student *j;
if(start==NULL)
{
start=t;
end=t;
}
else
{
j=start;
while(j!=NULL)
{
if(j->rollNumber>t->rollNumber)
{
break;
}
j=j->next;
}
if(j==NULL)
{
end->next=t;
t->prev=end;
end=t;
}
else
{
if(j==start)
{
t->next=start;
start->prev=t;
start=t;
}
else
{
t->next=j;
t->prev=j->prev;
j->prev->next=t;
j->prev=t;
}
}
}
```

```
}
void addNameIndexNode(struct IndexNode *t)
{
struct IndexNode *j;
if(startOfNameIndex==NULL)
{
startOfNameIndex=t;
endOfNameIndex=t;
}
else
{
j=startOfNameIndex;
while(j!=NULL)
{
if(strcmp(j->student->name,t->student->name)>0) break;
j=j->next;
}
if(j==NULL)
{
endOfNameIndex->next=t;
t->prev=endOfNameIndex;
endOfNameIndex=t;
}
else
{
if(j==startOfNameIndex)
{
t->next=startOfNameIndex;
startOfNameIndex->prev=t;
startOfNameIndex=t;
}
else
{
t->next=j;
t->prev=j->prev;
j->prev->next=t;
j->prev=t;
}
}
}
}

struct Student * exists(int rollNumber)
{
struct Student *t;
t=start;
while(t!=NULL)
{
if(t->rollNumber==rollNumber) break;
if(t->rollNumber>rollNumber)
```

```
{
t=NULL;
break;
}
t=t->next;
}
return t;
}

struct IndexNode * getNameIndexNode(struct Student *t)
{
struct IndexNode *ti;
ti=startOfNameIndex;
while(ti!=NULL)
{
if(ti->student==t) break;
ti=ti->next;
}
return ti;
}

void  displayStudent()
{
int rollNumber;
struct Student *t;
printf("Enter roll number");
scanf("%d",&rollNumber);
if(rollNumber<=0)
{
printf("Invalid roll number\n");
return;
}
t=exists(rollNumber);
if(t==NULL) printf("Invalid roll number\n");
else
{
printf("Name %s\n",t->name);
printf("Age %s\n",t->age);
}
}

void addStudent()
{
int rollNumber;
struct Student *t;
struct IndexNode *ti;
printf("Enter roll number");
scanf("%d",&rollNumber);
fflush(stdin);
if(rollNumber<=0)
```

```
{
printf("Invalid roll number\n");
return;
}
t=exists(rollNumber);
if(t!=NULL)
{
printf("That roll number alloted to %s - Age %d\n",t->name,t->age);
return;
}
t=getNewNode();
t->rollNumber=rollNumber;
printf("Enter name : ");
gets(t->name);
fflush(stdin);
printf("Enter age : ");
scanf("%d",&t->age);
fflush(stdin);
ti=getNewIndexNode();
ti->student=t;
addStudentNode(t);
addNameIndexNode(ti);
printf("Student added\n");
}
void displayRollNumberOrderedList()
{
struct Student *t;
char orderChoice;
printf("Selected ordering type\n");
printf("1. Ascending\n");
printf("2. Descending\n");
printf("Enter your choice ");
scanf("%d",&orderChoice);
fflush(stdin);
if(orderChoice<1 || orderChoice>2)
{
printf("Invalid choice \n");
return;
}
if(orderChoice==1)
{
t=start;
while(t!=NULL)
{
printf("Roll number %d\n",t->rollNumber);
printf("Name %s\n",t->name);
printf("Age %d\n",t->age);
t=t->next;
}
}
```

```
else
{
t=end;
while(t!=NULL)
{
printf("Roll number %d\n",t->rollNumber);
printf("Name %s\n",t->name);
printf("Age %d\n",t->age);
t=t->prev;
}
}
}
void displayNameOrderedList()
{
struct IndexNode *t;
char orderChoice;
printf("Selected ordering type\n");
printf("1. Ascending\n");
printf("2. Descending\n");
printf("Enter your choice ");
scanf("%d",&orderChoice);
fflush(stdin);
if(orderChoice<1 || orderChoice>2)
{
printf("Invalid choice \n");
return;
}
if(orderChoice==1)
{
t=startOfNameIndex;
while(t!=NULL)
{
printf("Roll number %d\n",t->student->rollNumber);
printf("Name %s\n",t->student->name);
printf("Age %d\n",t->student->age);
t=t->next;
}
}
else
{
t=endOfNameIndex;
while(t!=NULL)
{
printf("Roll number %d\n",t->student->rollNumber);
printf("Name %s\n",t->student->name);
printf("Age %d\n",t->student->age);
t=t->prev;
}
}
}
```

```
void deleteStudent()
{
int rollNumber;
struct Student *t;
struct IndexNode *ti;
printf("Enter roll number of the student to delete ");
scanf("%d",&rollNumber);
fflush(stdin);
if(rollNumber<=0)
{
printf("Invalid roll number\n");
return;
}
t=exists(rollNumber);
if(t==NULL)
{
printf("Invalid roll number\n");
return;
}
if(t==start && t==end)
{
start=end=NULL;
}
else
{
if(t==start)
{
start=start->next;
start->prev=NULL;
}
else
{
if(t==end)
{
end=end->prev;
end->next=NULL;
}
else
{
t->prev->next=t->next;
t->next->prev=t->prev;
}
}
}
ti=getNameIndexNode(t);
if(ti==startOfNameIndex && ti==endOfNameIndex)
{
startOfNameIndex=endOfNameIndex=NULL;
}
```

```
else
{
if(ti==startOfNameIndex)
{
startOfNameIndex=startOfNameIndex->next;
startOfNameIndex->prev=NULL;
}
else
{
if(ti==endOfNameIndex)
{
endOfNameIndex=endOfNameIndex->prev;
endOfNameIndex->next=NULL;
}
else
{
ti->prev->next=ti->next;
ti->next->prev=ti->prev;
}
}
}
printf("Student : %s with age %d deleted \n",t->name,t->age);
free(ti);
free(t);

}
void editStudent()
{
struct Student *t;
struct IndexNode *ti;
int rollNumber;
printf("Enter roll number of the student to edit");
scanf("%d",&rollNumber);
fflush(stdin);
if(rollNumber<=0)
{
printf("Invalid roll number\n");
return;
}
t=exists(rollNumber);
if(t==NULL)
{
printf("Invalid roll number\n");
return;
}
printf("Name %s\n",t->name);
printf("Enter new name");
gets(t->name);
fflush(stdin);
printf("Age %d\n",t->age);
```

```
printf("Enter new age");
scanf("%d",&t->age);
fflush(stdin);
ti=getNameIndexNode(t);
if(ti==startOfNameIndex && ti==endOfNameIndex)
{
startOfNameIndex=endOfNameIndex=NULL;
}
else
{
if(ti==startOfNameIndex)
{
startOfNameIndex=startOfNameIndex->next;
startOfNameIndex->prev=NULL;
}
else
{
if(ti==endOfNameIndex)
{
endOfNameIndex=endOfNameIndex->prev;
endOfNameIndex->next=NULL;
}
else
{
ti->prev->next=ti->next;
ti->next->prev=ti->prev;
}
}
}
ti->next=NULL;
ti->prev=NULL;
addNameIndexNode(ti);
printf("Student updated\n");
}


int main()
{
int ch;
while(1)
{
printf("1. Add Student\n");
printf("2. Display Student\n");
printf("3. Display List (Roll number ordered)\n");
printf("4. Display List (Name ordered)\n");
printf("5. Edit Student\n");
printf("6. Delete Student\n");
printf("7. Exit\n");
printf("Enter your choice ");
scanf("%d",&ch);
```

```
fflush(stdin);
if(ch==1) addStudent();
if(ch==2) displayStudent();
if(ch==3) displayRollNumberOrderedList();
if(ch==4) displayNameOrderedList();
if(ch==5) editStudent();
if(ch==6) deleteStudent();
if(ch==7) break;
}
return 0;
}

/*
```

Assignment : Make necessary changes and add age ordered index.

```
*/
```