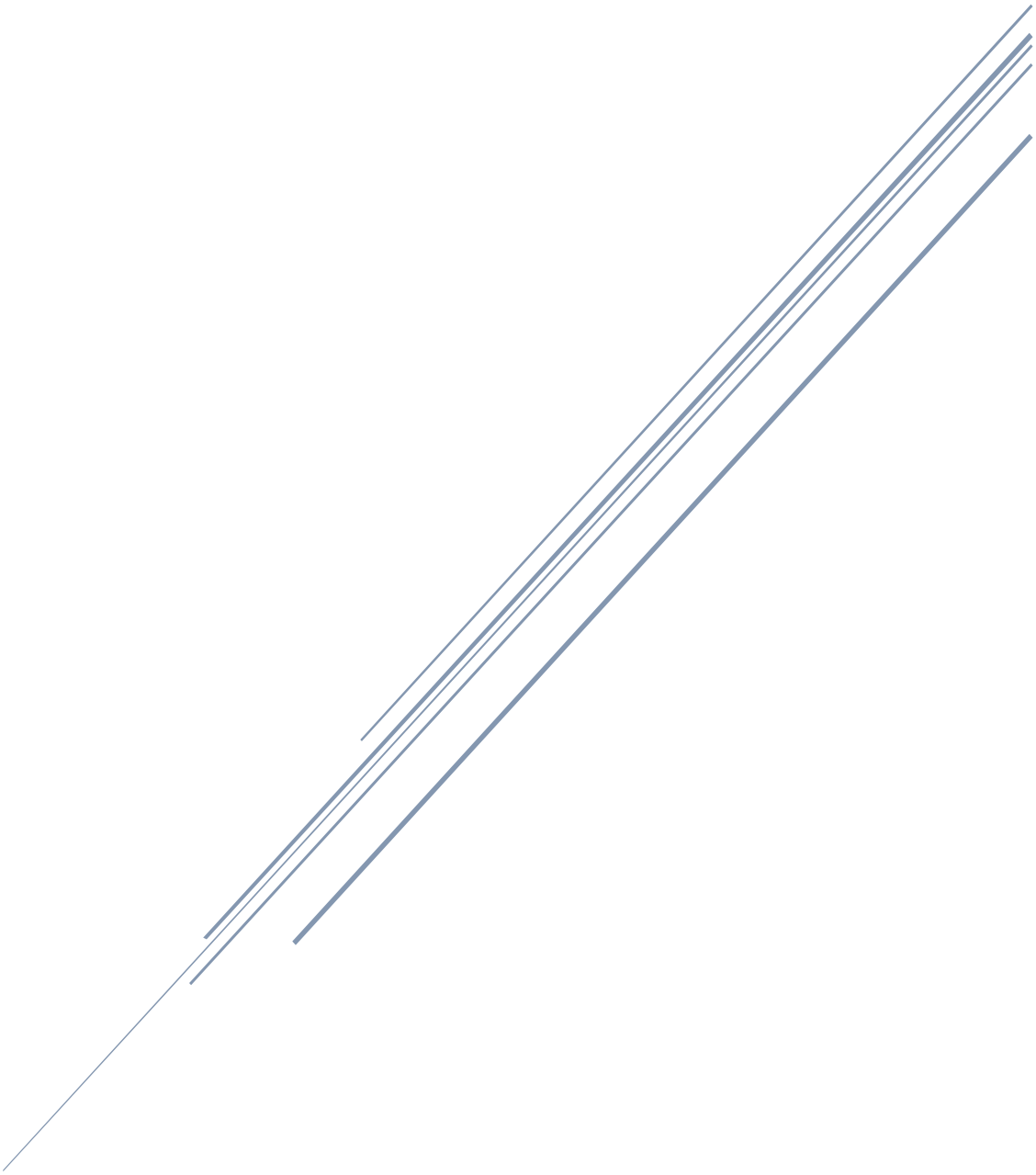
LINKED LISTS, STACKS AND QUEUES

Chapter 02 – Activity Book
Data Structures and Algorithms



Prakash Hegade
School of CSE, KLE TECH

1. Program to convert Infix expression to postfix expression

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

#define MAXCOLS 80
#define TRUE 1
#define FALSE 0

struct stack {
   int top;
   char operators[MAXCOLS];
};

int empty(struct stack *s);
int full(struct stack *s);
char peek(struct stack *s);
void push(struct stack *s, char item);
char pop(struct stack *s);
int prcd(char, char);
int priority(char op);

int main() {

   char infix[50], postfix[50], ch, topsymb;
   int i, j = 0;

   struct stack s;
   s.top = -1;

   printf("Enter a valid infix expression:\n");
   scanf("%s", infix);

   for (i = 0; infix[i] != '\0'; i++) {
      ch = infix[i];
      if (isdigit(ch) || isalpha(ch))
         postfix[j++] = ch;
      else {
         while (!empty(&s) && prcd(peek(&s), ch)) {
            topsymb = pop(&s);
            postfix[j++] = topsymb;
         }
         if(ch!=')')
            push(&s, ch);
         else
            topsymb = pop(&s);
      }
   }
```

```
   while (!empty(&s)) {
      topsymb = pop(&s);
      postfix[j++] = topsymb;
   }

   postfix[j] = '\0';

   printf("%s%s\n", "The infix expression is  ", infix);
   printf("%s%s\n", "The postfix expression is: ", postfix);

   return 0;
}

int empty(struct stack *ps) {
   if (ps->top == -1)
      return TRUE;
   return FALSE;
}

int full(struct stack *ps) {
   if (ps->top == MAXCOLS - 1)
      return TRUE;
   return FALSE;
}

char peek(struct stack *ps) {
   return ps->operators[ps->top];
}

int prcd(char op1, char op2) {
   if (op1 == '(' && op2 == ')')
      return FALSE;
   if (op1 == '(')
      return FALSE;
   if (op2 == '(' && op1 != ')')
      return FALSE;
   if (op2 == ')' && op1 != '(')
      return TRUE;
   if (priority(op1) == priority(op2) && priority(op2) == 3)
      return FALSE;
   if (priority(op1) >= priority(op2))
      return TRUE;
   return FALSE;
}

int priority(char op) {
   int precedence;
```

```
   switch (op) {
      case '$':precedence = 3;
         break;
      case '*':
      case '/':precedence = 2;
         break;
      case '+':
      case '-':precedence = 1;
         break;
      case ')':
      case '(':precedence = 0;
         break;
   }
   return precedence;
}

void push(struct stack *ps, char item) {
   if (full(ps)) {
      printf("%s\n", "STACK OVERFLOW");
      exit(1);
   }
   ps->operators[++(ps->top)] = item;
}

char pop(struct stack *ps) {
   if (empty(ps)) {
      printf("%s\n", "STACK UNDERFLOW");
      exit(1);
   }
   return ps->operators[(ps->top)--];
}
```

2. Evaluating a Postfix Expression

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define MAXCOLS 80
#define TRUE    1
#define FALSE   0

struct stack
{
   int top;
   double operands[MAXCOLS];
};
```

```
int full(struct stack *);
int empty(struct stack *);
void push(struct stack *, double);
double pop(struct stack *);
double oper(int, double, double);

int main()
{
   int i=0;
   double opnd1, opnd2, value;
   char expr[MAXCOLS];
   int ch;
   struct stack s;
   s.top=-1;

   printf("Enter a postfix expression to evaluate\n");
   scanf("%s", expr);

   for(i=0; expr[i]!='\0'; i++)
   {
      ch=expr[i];
      if(isdigit(ch))
      {
         // If It is an operand, push it inside the stack
         push(&s, (double) (ch - '0'));
      }
      else
      {
         // If is an operator pop 2 operands from stack and evaluate and push back the value
         opnd2=pop(&s);
         opnd1=pop(&s);
         value= oper(ch,opnd1,opnd2);
         push(&s, value);
      }
   }
   value= pop(&s);
   printf("value of expression is : %f\n\n", value);

   return 0;
}

int full(struct stack *ps)
{
      if(ps->top==MAXCOLS-1)
         return TRUE;
      else
         return FALSE;
}
```

```
int empty(struct stack *ps)
{
      if(ps->top==-1)
         return TRUE;
      else
         return FALSE;
}

void push(struct stack *ps, double x) {
      ps->operands[++(ps->top)] = x;
}

double pop(struct stack *ps) {
      double x;
      x = ps->operands[(ps->top)--];
      return x;
}

double oper(int symb, double opnd1, double opnd2)
{
      switch(symb)
      {
            case '+': return (opnd1 + opnd2);
            case '-': return (opnd1 - opnd2);
            case '*': return (opnd1 * opnd2);
            case '/': return (opnd1 / opnd2);
            case '$': return (double)pow(opnd1,opnd2);
            default : printf("%s\n", "ILLEGAL OPERATION");
                  exit(1);
      }
}
```

3. Program to check valid parenthesis in an expression

```
#include<stdio.h>
#include<string.h>
#include<stdlib.h>

#define MAX 50
int top=-1;
int stack[MAX];

void push(char);
char pop();
int match(char a,char b);
int check(char []);

int main()
{
```

```
	char exp[MAX];
	int valid;
	printf("Enter an algebraic expression : ");
	gets(exp);
	valid=check(exp);
	if(valid==1)
		printf("Valid expression\n");
	else
		printf("Invalid expression\n");

		return 0;

}

int check(char exp[] )
{
	int i;
	char temp;
	for(i=0;i<strlen(exp);i++)
	{
		if(exp[i]=='(' || exp[i]=='{' || exp[i]=='[')
			push(exp[i]);
		if(exp[i]==')' || exp[i]=='}' || exp[i]==']')
			if(top==-1)
			{
				printf("Right parentheses are more than left parentheses\n");
				return 0;
			}
			else
			{
				temp=pop();
				if(!match(temp, exp[i]))
				{
					printf("Mismatched parentheses are : ");
					printf("%c and %c\n",temp,exp[i]);
					return 0;
				}
			}
	}
	if(top==-1) {
		printf("Balanced Parentheses\n");
		return 1;
	}
	else {
		printf("Left parentheses more than right parentheses\n");
		return 0;
	}
}
```

```
int match(char a, char b)
{
        if(a=='[' && b==']')
                return 1;
        if(a=='{' && b=='}')
                return 1;
        if(a=='(' && b==')')
                return 1;
        return 0;
}

void push(char item) {
        if(top==(MAX-1)) {
                printf("Stack Overflow\n");
                return;
        }
        top=top+1;
        stack[top]=item;
}

char pop() {
        if(top==-1) {
                printf("Stack Underflow\n");
                exit(1);
        }
        return(stack[top--]);
}
```

Priority Queue

Priority Queue is a data structure in which the intrinsic ordering of the elements does determine the results of its basic operations.

Ascending priority Queue:

- Is collection of items into which items can be inserted arbitrarily and from which only smallest item can be removed.

Descending priority Queue:

- Is collection of items into which items can be inserted arbitrarily and from which only largest item can be removed.

A priority queue always has an **"order by"** parameter. On what basis do we prioritize in the queue is the order by parameter. It can be machine generated like unique number or given by user like first name, last name, telephone number or anything depending on application or usage.

How to implement a priority Queue:

There are several solutions and each has its own merits and demerits. Either insertion or deletion becomes inefficient.

1. A special empty indicator can be placed at deleted position. Insertion proceeds as before. If there is no room for new insertion compaction takes places. The deleted values are cleared and a space is made for new coming entries.

2. A special empty indicator can be placed at deleted position but while insertion we insert at first available free space.

3. After every deletion compaction takes place. Insertion remains unchanged. On an average half of all priority queue elements are shifted for every deletion. Shifting of smaller group can be done to reduce the number of shifts.

4. Maintain queue as circular array and deletion operation is unchanged. Every insertion happens at rightful position (Space is made available and insertion happens in sorted fashion). We maintain an ordered array.

4. Linear Descending Priority Queue Implementation

```
#include <stdio.h>
#include <stdlib.h>
#define MAXQUEUE 5
#define TRUE   1
#define FALSE   0

struct PQUEUE
{
   int front;
   int rear;
   int items[MAXQUEUE];
};
typedef struct PQUEUE PQUEUE;

void Enqueue(PQUEUE *, int, int *);
void Dequeue(PQUEUE *, int *, int *);
void compaction(PQUEUE *);
void Display(PQUEUE *);
int empty(PQUEUE *);
int full(PQUEUE *);

int main()
{
      int choice =0, x;
      int overflow = 0, underflow = 0;
      PQUEUE q;
      q.front=0;
      q.rear=-1;
```

```
    while(1)
    {
      printf("\n**** MENU ****\n");
      printf("1 - Enqueue\n");
      printf("2 - Dequeue\n");
      printf("3 - Display\n");
      printf("4 - Exit\n");
      printf("**************\n");
      printf("Enter your choice: ");
      scanf("%d", &choice);

        switch(choice)
        {
          case 1: printf("Enter item to insert:\t");
                  scanf("%d", &x);
                  Enqueue(&q, x, &overflow);
                  if(overflow)
                    printf("PQUEUE FULL. Insertion Failed!");
                  else
                    printf("Item %d inserted into PQUEUE rear successfully\n", x);
                  break;

          case 2: Dequeue(&q, &x, &underflow);
                  if(underflow)
                    printf("PQUEUE EMPTY. Deletion cannot be performed\n");
                  else
                    printf("Item %d is removed from front of the PQUEUE\n", x);
                  break;

          case 3: if(empty(&q))
                    printf("PQUEUE EMPTY\n");
                  else
                    Display(&q);
                  break;

          case 4: printf("Thank you. Program will Exit. \n");
                  exit(0);
        }
    }
    return 0;
}

void Enqueue(PQUEUE *pq, int x, int *poverflow)
{
  // If full then call compaction
  if(full(pq))
  {
    compaction(pq);
```

```
    // If still full, then no more items can be added
    if(full(pq))
    {
      *poverflow=TRUE;
      return;
    }
  }
  *poverflow=FALSE;
  (pq->rear) ++;
  pq->items[pq->rear]=x;
}


// Compaction function will make space by shifting down with deleted values
void compaction(PQUEUE *pq)
{
  int i = 0, j = 0;
  for (i = 0; i <pq->rear; i++)
  {
    // It there is a deleted value, shift all by one down and decrement the rear
    if(pq->items[i] == -1)
    {
      for(j = i; j< pq->rear; j++)
      {
        pq->items[j] = pq->items[j+1];
      }
      (pq->rear)--;
    }
    // is shifted values again the deleted one? if yes continue the process with same index
    if(pq->items[i] == -1)
        i--;
  }
  // if the last item pointed by rear is deleted
  if(pq->items[pq->rear] == -1)
    (pq->rear)--;
}

void Dequeue(PQUEUE *pq, int *px, int *punderflow)
{
  if(empty(pq))
    *punderflow=TRUE;
  else
  {
     *punderflow=FALSE;
     int dequeue_data = 0;
     int dequeue_index= 0;
     int i =0;
     dequeue_data = pq->items[pq->front];
```

```
      dequeue_index = 0;

    // Find the largest in the queue
    for(i = 1; i <= pq->rear; i++)
    {
      if(pq->items[i] > dequeue_data)
      {
        dequeue_data = pq->items[i];
        dequeue_index = i;
      }
    }

    // If all are deleted values, reset the queue
    if(dequeue_data == -1)
    {
        *punderflow = TRUE;
        pq->front = 0;
        pq->rear = -1;
        return;
    }

    // Return the largest to user and delete it
    *px = pq->items[dequeue_index];
    pq->items[dequeue_index] = -1;
  }
}

// Front will always be pointing to index 0, hence the empty condition
int empty(PQUEUE *pq)
{
  if(pq->front == 0 && pq->rear == -1)
    return TRUE;
  else
    return FALSE;
}

// Full condition will remain the same as linear queue
int full(PQUEUE *pq)
{
  if(pq->rear == MAXQUEUE-1)
    return TRUE;
  else
    return FALSE;
}
```

Note: Display() is same as the linear queue. While displaying skip if the data is -1

5. Linear Ascending Priority Queue Implementation

There will be change in compaction and dequeue function which is as given below. Rest of the implementation will remain the same.

```
// Compaction function will make space by shifting down with deleted values
void compaction(PQUEUE *pq)
{
   int i = 0, j = 0;
   for (i = 0; i <pq->rear; i++)
   {
      // It there is a deleted value, shift all by one down and decrement the rear
      if(pq->items[i] == -1)
      {
         for(j = i; j< pq->rear; j++)
            pq->items[j] = pq->items[j+1];

         (pq->rear)--;
      }
   }
   // if the last item pointed by rear is deleted
   if(pq->items[pq->rear] == -1)
      (pq->rear)--;
}

void Dequeue(PQUEUE *pq, int *px, int *punderflow)
{
   *punderflow = TRUE;
   if(empty(pq))
         return;
   else
   {
      // Call the compaction before every deletion
      compaction(pq);
      if(empty(pq))
         return;

      *punderflow=FALSE;
      int dequeue_data = 0, dequeue_index= 0, i = 0;
      dequeue_data = pq->items[dequeue_index];

     // Find the smallest in the queue
      for(i = 1; i <= pq->rear; i++)
      {
         if(pq->items[i] < dequeue_data)
         {
            dequeue_data = pq->items[i];
            dequeue_index = i;
         }
```

```
    }
    // Return the smallest to user and delete it
    *px = pq->items[dequeue_index];
    pq->items[dequeue_index] = -1;
  }
}
```

6. Doubly Linked List Insert and Delete at Positions

```
NODE * insert_at_position(NODE * start)
{
  NODE *newnode, *tempnode;
  int position=0, i=0;

  printf("Enter insert position\n");
  scanf("%d", &position);
  if(position < 1 || position > currnodes+1 )
    printf("Invalid position\n");
  else
  {
    newnode = getnode();
    if(newnode == NULL)
      return start;
    getdata(newnode);

    if(start == NULL)
      start = newnode;
    else if(position == 1)
    {
      newnode->next= start;
      start->prev=newnode;
      start = newnode;
    }
    else if(position == currnodes+1)
    {
      tempnode = start;
      while(tempnode->next != NULL)
        tempnode = tempnode->next;

      tempnode->next = newnode;
      newnode->prev = tempnode;
    }
    else
    {
      tempnode = start;
      for(i=2;i<position;i++)
        tempnode = tempnode->next;
```

```
        newnode->next = tempnode->next;
        newnode->prev =  tempnode;
        tempnode->next->prev = newnode;
        tempnode->next = newnode;
      }
      currnodes++;
      printf("%d info is inserted at the %d position of the doubly linked list\n", newnode->info,
position);
   }
   return start;
}

NODE * delete_from_position(NODE * start)
{
   NODE * tempnode, *prevnode;
   int position=0, i=0;

   if(start == NULL)
      printf("List is empty\n");
   else
   {
   printf("Enter delete position\n");
   scanf("%d", &position);
   if(position < 1 || position > currnodes )
     printf("Invalid position\n");
   else
   {
      if(currnodes == 1)
      {
         tempnode = start;
         start = NULL;
      }
      else if(position == 1)
      {
         tempnode = start;
         start = start->next;
         start->prev = NULL;
      }
      else if(position == currnodes)
      {
         tempnode = start;
         while(tempnode->next != NULL)
            tempnode = tempnode->next;

         prevnode = tempnode;
         prevnode = prevnode->prev;
         prevnode->next = NULL;
```

```
      }
      else
      {
         prevnode = NULL;
         tempnode = start;
         i = 2;
         while(i<=position)
         {
            prevnode = tempnode;
            tempnode = tempnode->next;
            i++;
         }
         prevnode->next = tempnode->next;
         tempnode->next->prev = prevnode;
      }
      printf("Node %d deleted from the %d position from the Doubly linked list\n", tempnode->info,
position);
      free(tempnode);
      currnodes--;
   }
 }
   return start;
}
```

8. Program to add two polynomials

```
#include <stdio.h>
#include <stdlib.h>
struct node
{
   int coeff;
   int x;
   int y;
   int flag;
   struct node * next;
};
typedef struct node NODE;

NODE * getnode()
{
   NODE * newnode;
   newnode = (NODE *)malloc (sizeof(NODE));
   if (newnode == NULL)
   {
      printf("Memory allocation failed\n");
      exit(0);
   }
   return newnode;
}
```

```
void display (NODE * head)
{
   NODE * temp;
   if(head->next == head)
      printf("Polynomial does not exist\n");

   for (temp = head->next; temp!=head; temp = temp->next)
      printf(" + %d x ^ %d y ^ %d ", temp->coeff, temp->x, temp->y);
}

NODE * insert_at_end( int coeff, int x, int y, NODE * head)
{
   NODE * temp, *curr;
   temp = getnode();
   temp->coeff = coeff;
   temp->x = x;
   temp->y = y;
   temp->flag = 0;

   curr = head->next;
   while(curr->next != head)
      curr = curr->next;

   curr->next = temp;
   temp->next = head;
   return head;
}

NODE * read_polynomial(NODE * head)
{
   int coeff, x, y;
   int choice;
   while(1)
   {
      printf("Enter the Coefficient\n");
      scanf("%d", &coeff);
      printf("Enter power of x\n");
      scanf("%d", &x);
      printf("Enter power of y\n");
      scanf("%d", &y);

      head =  insert_at_end(coeff, x, y, head);
      printf("Do you want to continue??\n");
      printf("1-yes\n2-0\n");
      scanf("%d", &choice);
      if(choice == 0)
         break;
```

```
   }
   return head;
}

NODE * add_polynomial(NODE * head1, NODE * head2, NODE * head3)
{
     NODE * p1, *p2;
     int coeff1, coeff2, x1, x2,y1, y2, coeff;

     p1 = head1->next;
     // For every node in the linked list for first polynomial
     while (p1 != head1)
     {
       // Get the coeff, x power and y power
       coeff1 = p1->coeff;
       x1 = p1->x;
       y1 = p1->y;

       // Find a match for it in linked list of second polynomial
       p2 = head2->next;
       while(p2 != head2)
       {
          coeff2 = p2->coeff;
          x2 = p2->x;
          y2 = p2->y;

          // If there is a match, break
          if(x1 == x2 && y1 == y2)
             break;

          // Else Advance to the next node of second polynomial
          p2 = p2->next;
       }

       // If match was found in second polynomial
       if(p2 != head2)
       {
          // Add the coefficients
          coeff = coeff1 + coeff2;

          // Set the flag to traversed
          p2->flag = 1;

          // If the coeff is not 0, add it to resulting list
          if(coeff != 0)
             head3 = insert_at_end(coeff, x1, y1, head3);
       }
       // Else insert the node from first polynomial to the linked list
```

```
        else
           head3 = insert_at_end(coeff1, x1, y1, head3);

        // Advance the pointer of first polynomial
        p1 = p1->next;
      }

      // Add all the unvisited nodes of second polynomial to the resulting list
      p2 = head2->next;
      while(p2 != head2)
      {
        if(p2->flag == 0)
           head3 = insert_at_end(p2->coeff, p2->x, p2->y, head3);
         p2= p2->next;
      }
     return head3;
}

int main()
{
   NODE * head1, *head2, *head3;
   head1 = getnode();
   head2 = getnode();
   head3 = getnode();

   head1->next = head1;
   head2->next = head2;
   head3->next = head3;

   printf("%s\n","Enter the first polynomial");
   head1 = read_polynomial(head1);

   printf("%s\n","Enter the second polynomial");
   head2 = read_polynomial(head2);

   head3 = add_polynomial(head1, head2, head3);

   printf("%s", "\nThe first polynomial is..\n");
   display(head1);

   printf("%s", "\nThe second polynomial is..\n");
   display(head2);

   printf("%s", "\nThe sum of two polynomial is..\n");
   display(head3);

   return 0;
}
```

9. Program to add two long integers

```
#include <stdio.h>
#include <stdlib.h>
struct node
{
   int info;
   struct node * next;
};
typedef struct node NODE;

NODE * getnode()
{
   NODE * newnode;
   newnode = (NODE *)malloc (sizeof(NODE));
   if(newnode == NULL)
   {
      printf("Memory Allocation Failed\n");
      exit(0);
   }
   return newnode;
}

/*
The two number input by user will be from MSB to LSB.
say 4341 and
     2334 are two input numbers.

We need to add the number from LSB. If LSB resides at the end of the list, then it becomes difficult
to traverse the list and then add from end of the list. Instead we use the function
insert_at_start().
So that LSB resides at the start of the list.
*/
NODE * insert_at_start(int item, NODE * head)
{
   NODE * temp;

   temp = getnode();
   temp->info = item;

   temp->next = head->next;
   head->next = temp;

   // header keeps the number of nodes present in the list
   head->info = head->info + 1;

   return head;
}
```

```
/*
The number when added from LSB,
 1434 and
 4332
 --------
 5766 --> result obtained is from the LSB to MSB.
So, we use the function insert_at_end()
*/
NODE * insert_at_end(int item, NODE * head)
{
   NODE * temp, *curr;

   temp = getnode();
   temp->info = item;

   curr = head->next;
   while(curr->next != head)
      curr = curr->next;

   curr->next = temp;
   temp->next = head;

   head->info =  head->info + 1;
   return head;
}


NODE * read_number(NODE * head)
{
   int item = 0;
   printf("Enter the number digit wise [Hit enter key after every input]. Enter 999 to end input\n");
   while(1)
   {
      scanf("%d", &item);
      if(item == 999)
         break;
      head = insert_at_start(item, head);
   }
   return head;
}

void display_number(NODE * head)
{
   NODE * curr;
   int * a;
   int i, j;

   if(head->next == head)
```

```
  {
    printf("Number not entered\n");
    return;
  }
  i = head->info;
  a = (int *) malloc(i * sizeof(int));
  // First copy the information into the array
  for( curr = head->next, j = 0; curr != head; curr = curr->next)
    a[j++] = curr->info;

  // print it from last to first
  while(--j != -1)
    printf("%d", a[j]);
  printf("\n");
}

NODE * add_long(NODE * head1, NODE * head2, NODE * head3)
{
  NODE * list, *list1, *list2, *head;
  int sum, carry, digit;
  carry = 0;

  // Initialize two pointers each with a linked list carrying numbers to be added
  list1 = head1->next;
  list2 = head2->next;

  // while we have still not reached the end of both the lists
  while(list1 != head1 && list2 != head2)
  {
    // Add the info of from both the lists
    sum = list1->info + list2->info + carry;

    // Separate out LSB and MSB to obtain carry
    digit = sum % 10;
    carry = sum / 10;

    // Insert the digit into the resulting list
    head3 = insert_at_end(digit, head3);

    // Update both pointers
    list1 = list1->next;
    list2 = list2->next;
  }

  // Are there any more nodes remaining in any of the list?
  // If list1 is not yet exhausted,
  if(list1 != head1)
  {
```

```
      list =  list1;
      head =  head1;
   }
   // Or if list2 is not yet exhausted
   else
   {
      list = list2;
      head = head2;
   }

   // For all the remaining nodes in the list
   while (list != head)
   {
      // Get the sum
      sum  = list->info + carry;

       // Separate out LSB and MSB to obtain carry
      digit = sum % 10;
      carry = sum / 10;

       // Insert the digit into the resulting list
      head3 = insert_at_end(digit, head3);

      // Update the pointer
      list = list->next;
   }

   // If we still remain with carry, insert it into the resulting list
   if(carry == 1)
      head3 = insert_at_end(carry, head3);

   // return the head of resulting list
   return head3;
}

int main()
{
   NODE * head1, *head2, *head3;

   head1 = getnode();
   head2 = getnode();
   head3 = getnode();
   head1->next = head1;
   head2->next = head2;
   head3->next = head3;

   head1->info = head2->info = head3->info = 0;
```

```
    printf("Enter the first number\n");
    head1 = read_number(head1);

    printf("Enter the second number\n");
    head2 = read_number(head2);

    head3 =  add_long(head1, head2, head3);

    printf("Number 01 is \n");
    display_number(head1);

    printf("Number 02 is \n");
    display_number(head2);

    printf("Sum is \n");
    display_number(head3);
    return 0;
}
```

~*~*~*~*~*~*~*~