1. **Reverse a singly-linked linked-list**

// iterative version

Node\* ReverseList(Node \*\* List)

{

Node \* temp1 = \*List;

Node \* temp2 = NULL;

Node \* temp3 = NULL;

while (temp1)

{

\*List = temp1; // set the head to last node

temp2 = temp1->pNext; // save the next ptr in temp2

temp1->pNext = temp3; // change next to previous

temp3 = temp1;

temp1 = temp2;

}

return \*List;

}

1. **Delete a node in doubly-linked linked-list**

void deleteNode(node \*n)

{

node \*np = n->prev;

node \*nn = n->next;

np->next = n->next;

nn->prev = n->prev;

delete n;

}

1. **Reverse a string**

void ReverseString (char \*String)

{

char \*Begin = String;

char \*End = String + strlen(String) - 1;

char TempChar = '\0';

while (Begin < End)

{

TempChar = \*Begin;

\*Begin = \*End;

\*End = TempChar;

Begin++;

End--;

}

}

1. **Sort a singly-linked list**

// sorting in descending order

struct node

{

int value;

node\* NEXT;

}

// Assume Head pointer denotes the first element in the linked list;

// only need to change the values, not the pointers

Sort(Node \*Head)

{

node\* first, second, temp;

first = Head;

while(first != null)

{

second = first->NEXT;

while(second!=null)

{

if(first->value < second->value)

{

temp = new node();

temp->value=first->value;

first->value=second->value;

second->value=temp->value;

delete temp;

}

second = second->NEXT;

}

first = first->NEXT;

}

}

1. **Insert a node in a sorted linked list**

void sortedInsert(Node \* head, Node \* newNode)

{

Node \* current = head;

// traverse list until you find item bigger than new node value

while (current != NULL && current->data < newNode->data)

{

current = current->next;

}

// insert the new node before the big item

newNode->next = current->next;

current = newNode;

}

1. **Convert string to upper case**

void ToUpper(char \* S)

{

while (\*S != 0)

{

\*S = (\*S >= 'a' && \*S <= 'z') ? (\*S-'a'+'A') : \*S;

S++;

}

}

1. **Multiply a number by 7 without using \* and + operator.**

NewNum = Num << 3; // mulitplied by 2 ^ 3 = 8

NewNum = NewNum - Num; // 8 – 1 = 7

1. **Write a function that takes in a string parameter and checks to see whether or not it is an integer, and if it is then return the integer value.**

#include <stdio.h>

int strtoint(char \*s)

{

int index = 0, flag = 0;

while( \*(s+index) != '\0')

{

if( (\*(s + index) >= '0') && \*(s + index) <= '9')

{

flag = 1;

index++;

}

else

{

flag = 0;

break;

}

}

if( flag == 1 )

return atoi(s);

else

return 0;

}

main()

{

printf("%d", strtoint("0123"));

printf("\n%d", strtoint("0123ii"));

}

1. **Print data from a binary tree – In-order(ascending)**

// recursive version

Void PrintTree (struct \* node node)

{

if ( node == NULL )

return;

PrintTree(node->left);

Printf(“%d”, node->data);

PrintTree(node->right);

}

1. **Print integer using only putchar function**

// recursive version

void PrintNum (int Num)

{

if (Num == 0)

return;

PrintNum (Num / 10);

Puthcar (‘0’ + Num % 10);

}

1. **Find the factorial of number**

// recursive version

int Factorial( int Num )

{

if(num > 0)

return Num \* Factorial (Num –1);

else

return 1;

}

// iterative version

int Factorial( int Num )

{

int i;

int result = 1;

for (i = Num; i > 0; i-- )

{

result = result \* i;

}

return result;

}

1. **Generate Fib numbers:**

int fib( n ) // recursive version

{

if ( n < 2 )

return 1;

else

return fib ( n –1 ) + fib ( n –2 );

}

int fib( n ) //iterative version

{

int f1 = 1, f2 = 1;

if ( n < 2 )

return 1;

for ( i = 1; i < N; i++)

{

f = f1 + f2;

f1= f2;

f = f1;

}

return f;

}

1. **Write a function that finds the last instance of a character in a string**

char \*lastchar(char \*String, char ch)

{

char \* pStr = NULL;

// traverse the entire string

while(\*String++ != NULL)

if( \*String == ch )

pStr = String;

return pStr;

}

1. **Return the Nth node from the end of the linked list in one pass.**

Node \* GetNthNode (Node\* Head, int NthNode)

{

Node \* pNthNode = NULL;

Node \* pTempNode = NULL;

int nCurrentElement = 0;

for (pTempNode = Head; pTempNode != NULL;

pTempNode = pTempNode->pNext)

{

nCurrentElement++;

if (nCurrentElement - NthNode == 0)

{

pNthNode = Head;

}

else if(nCurrentElement - NthNode > 0)

{

pNthNode = pNthNode ->pNext;

}

}

if (pNthNode )

return pNthNode;

else

return NULL;

}

1. **Counting set bits in a number.**

// First version:

int CoutSetBits(int Num)

{

for(int count=0; Num; Num >>= 1)

if (Num & 1)

count++;

return count;

}

// Optimized version:

int CoutSetBits(int Num)

{

for(int count = 0; Num; count++)

Num &= Num - 1;

return count;

}

1. **Binary Search**

// this function returns the location of key in the list;

// if the value is not found returns -1

int BinarySearch(int list[], int size, int key)

{

int left, right, midpt;

left = 0;

right = size - 1;

while (left <= right)

{

midpt = (int) ((left + right) / 2);

if (key == list[midpt])

return midpt;

else if (key > list[midpt])

left = midpt + 1;

else

right = midpt - 1;

}

return -1;

}

// Recursive

int BinarySearch(int sortedArray[], int first, int last, int key)

{

if (first <= last)

{

// compute mid point

int mid = (first + last) / 2;

if (key == sortedArray[mid])

return mid; // found it

else if (key < sortedArray[mid])

// call itself for the lower part of the array

return BinarySearch2(sortedArray, first, mid-1, key);

else

// call itself for the upper part of the array

return BinarySearch2(sortedArray, mid+1, last, key);

}

return -1; // failed to find key

}

1. **State Machines**

State machines are very simple in C if you use function pointers.

Basically you need 2 arrays - one for state function pointers and one for state transition rules. Every state function returns the code, you lookup state transition table by state and return code to find the next state and then just execute it.

int entry\_state(void);  
int foo\_state(void);  
int bar\_state(void);  
int exit\_state(void);  
  
/\* array and enum below must be in sync! \*/  
int (\* state)(void)[] = { entry\_state, foo\_state, bar\_state, exit\_state};  
enum state\_codes { entry, foo, bar, end};  
  
enum ret\_codes { ok, fail, repeat};  
struct transition {  
    enum state\_codes src\_state;  
    enum ret\_codes   ret\_code;  
    enum state\_codes dst\_state;  
};  
/\* transitions from end state aren't needed \*/  
struct transition state\_transitions[] = {  
    {entry, ok,     foo},  
    {entry, fail,   end},  
    {foo,   ok,     bar},  
    {foo,   fail,   end},  
    {foo,   repeat, foo},  
    {bar,   ok,     end},  
    {bar,   fail,   end},  
    {bar,   repeat, foo}};  
  
#define EXIT\_STATE end  
#define ENTRY\_STATE entry  
  
int main(int argc, char \*argv[]) {  
    enum state\_codes cur\_state = ENTRY\_STATE;  
    enum ret\_codes rc;  
    int (\* state\_fun)(void);  
  
    for (;;) {  
        state\_fun = state[cur\_state];  
        rc = state\_fun();  
        if (EXIT\_STATE == cur\_state)  
            break;  
        cur\_state = lookup\_transitions(cur\_state, rc);  
    }  
  
    return EXIT\_SUCCESS;  
}

<http://stackoverflow.com/questions/1371460/state-machines-tutorials>

Links:

<http://rcamera.org/?c-strings.article>

<http://www.globalguideline.com/interview_questions/Questions.php?sc=Programming_Algorithms>

<http://www.linuxquestions.org/questions/programming-9/extract-substring-from-string-in-c-432620/>

Strings within strings:

<http://www.dreamincode.net/code/snippet527.htm>

<http://www.allinterview.com/Interview-Questions/Software.html>