

```

/**
 * listops.cpp:
 * Prozedurale Listenimplementation
 */
#include <iostream>
#include <iomanip>
using namespace std;

/**
 * Deklaration von Typen und Funktionen
 */

// Listenstruktur-Datentyp mit zwei Variablen
struct ListElem
{
    int data;                // Datenelement
    ListElem * pNext;       // Zeiger auf Nachfolger
};

// Element an eine Liste anhaengen
void list_append(ListElem *& pA, int to_append);
// Element in eine Liste einsortieren
void list_insert_sorted(ListElem *& pA, int to_insert);
// Element aus einer Liste loeschen
void list_delete(ListElem *& pA, int to_delete);
// Element in einer Liste suchen
ListElem * list_search(ListElem *& pA, int to_search);
// Liste komplett loeschen
void list_delete_all(ListElem *& pA);
// Liste ausgeben (max. 5 Elemente pro Zeile)
void list_print(ListElem *& pA);

/**
 * Definition von Funktionen
 */

void list_append(ListElem *& pA, int to_append)
{
    // 2 Faelle: Liste leer, Liste nicht leer
    if(pA == NULL){
        // 1. Fall: Anker auf neues erstes
        // Element setzen und Nachfolger auf NULL
        pA = new ListElem;
        pA -> data = to_append;
        pA -> pNext = NULL;
    } else {
        // 2. Fall: an das Ende der Liste gehen,
        // ein neues Element erzeugen und an das
        // Ende anhaengen
        ListElem * pLast = pA;
        while(pLast -> pNext != NULL)
            pLast = pLast -> pNext;
        ListElem * pNew = new ListElem;
        pNew -> data = to_append;
        pNew -> pNext = NULL;
        pLast -> pNext = pNew;
    }
}

```

```

void list_insert_sorted(ListElem *& pA, int to_insert)
{
    // 4 Faelle: Einfuegen am Anfang, am Ende,
    // in der Mitte oder in eine leere Liste

    // neues Element erzeugen
    ListElem * pNew = new ListElem;
    pNew -> data = to_insert;

    // 1. Fall: leere Liste
    if(pA == NULL){
        pA = pNew;
        pNew -> pNext = NULL;
    }
    // 2. Fall: Anfang der Liste
    else if(pA -> data >= to_insert){
        pNew -> pNext = pA;
        pA = pNew;
    } else {
        // korrekte Position in der Liste suchen
        ListElem * pSearch = pA;
        while(pSearch -> pNext != NULL &&
            pSearch -> pNext -> data < to_insert){
            pSearch = pSearch -> pNext;
        }

        // pSearch zeigt nun auf Element, vor welches
        // eingefuegt werden soll

        // 3. Fall: Ende der Liste
        if (pSearch -> pNext == NULL) {
            pNew -> pNext = NULL;
            pSearch -> pNext = pNew;
        }
        // 4. Fall: Mitte der Liste
        else {
            pNew -> pNext = pSearch -> pNext;
            pSearch -> pNext = pNew;
        }
    }
}

ListElem * list_search(ListElem *& pA, int to_search)
{
    // Element suchen und Zeiger darauf zurueckgeben
    ListElem * pSearch = pA;
    while(pSearch != NULL){
        if(pSearch -> data == to_search)
            return pSearch;
        pSearch = pSearch -> pNext;
    }
    return NULL;
}

```

```

void list_print(ListElem *& pA)
{
    cout << endl << endl;
    // Hilfszeiger zum Durchlaufer der Liste
    ListElem * pHelp = pA;
    // Zaehlervariable fuer Anzahl Elemente pro Zeile
    int count = 0;
    // solange durchlaufen, bis Ende erreicht
    while(pHelp != NULL){
        count ++;
        // setw(10) erzeugt 10 Zeichen breite Spalte
        // in der restliche Ausgabe steht
        cout << setw(10) << pHelp -> data;
        pHelp = pHelp -> pNext;
        // nach 5 Elementen einen Zeilenumbruch machen
        if(count == 5){
            cout << endl;
            count = 0;
        }
    }
    cout << endl << endl;
}

void list_delete(ListElem *& pA, int to_delete)
{
    // Hilfszeiger zum "Festhalten" des zu loeschenden Elementes
    ListElem * pDel;

    // 1. Fall: leere Liste
    if(pA == NULL)
        return;
    // 2. Fall: erstes Element zu loeschen
    if(pA -> data == to_delete){
        pDel = pA;
        pA = pA -> pNext;
        delete pDel;
    } else {
        // Vorgaenger des zu loeschenden Elements suchen
        ListElem * pSearch = pA;
        while( pSearch -> pNext != NULL &&
            pSearch -> pNext -> data != to_delete )
            pSearch = pSearch -> pNext;

        // pSearch zeigt nun auf Vorgaenger des zu
        // loeschenden Elements

        // 3. Fall: Element nicht gefunden
        if(pSearch -> pNext == NULL)
            return;
        // 4. Fall: mittleres oder letztes Element zu loeschen
        else {
            pDel = pSearch -> pNext;
            pSearch -> pNext = pSearch -> pNext -> pNext;
        }
    }
}

```

```

ListElem * list_search(ListElem *& pA, int to_search)
{
    ListElem * pSearch = pA;
    while(pSearch != NULL){
        if(pSearch -> data == to_search)
            return pSearch;
        pSearch = pSearch -> pNext;
    }
    return NULL;
}

void list_delete_all(ListElem *& pA)
{
    ListElem * pHelp = pA;
    while(pHelp != NULL){
        ListElem *pDel = pHelp;
        pHelp = pHelp -> pNext;
        delete pDel;
    }
    pA = NULL;
}

/**
 * Hauptprogrammfunktion
 */
int main()
{
    // Listenanker bereitstellen (leere Liste)
    ListElem * pA = NULL;
    // Element aus leerer Liste loeschen
    list_delete(pA,5);
    // in eine leere Liste einfuegen
    list_insert_sorted(pA,-2);
    // Elemente anhaengen
    for(int i=0; i<20; i+=2)
        list_append(pA,i);
    // am Anfang der Liste einfuegen
    list_insert_sorted(pA,-5);
    // am Ende der Liste einfuegen
    list_insert_sorted(pA,1000);
    // in der Mitte der Liste einfuegen
    list_insert_sorted(pA,5);
    // erstes Element loeschen
    list_delete(pA,-5);
    // letztes Element loeschen
    list_delete(pA,1000);
    // mittleres Element loeschen
    list_delete(pA,6);
    // nicht vorhandenes Element loeschen
    list_delete(pA,50);
    // Element suchen
    if(list_search(pA,5))
        cout << "Element 5 gefunden!" << endl;
    else
        cout << "Element 5 nicht gefunden!" << endl;
    // Liste ausgeben
    list_print(pA);
    // Liste loeschen
    list_delete_all(pA);
    return 0;
}

```