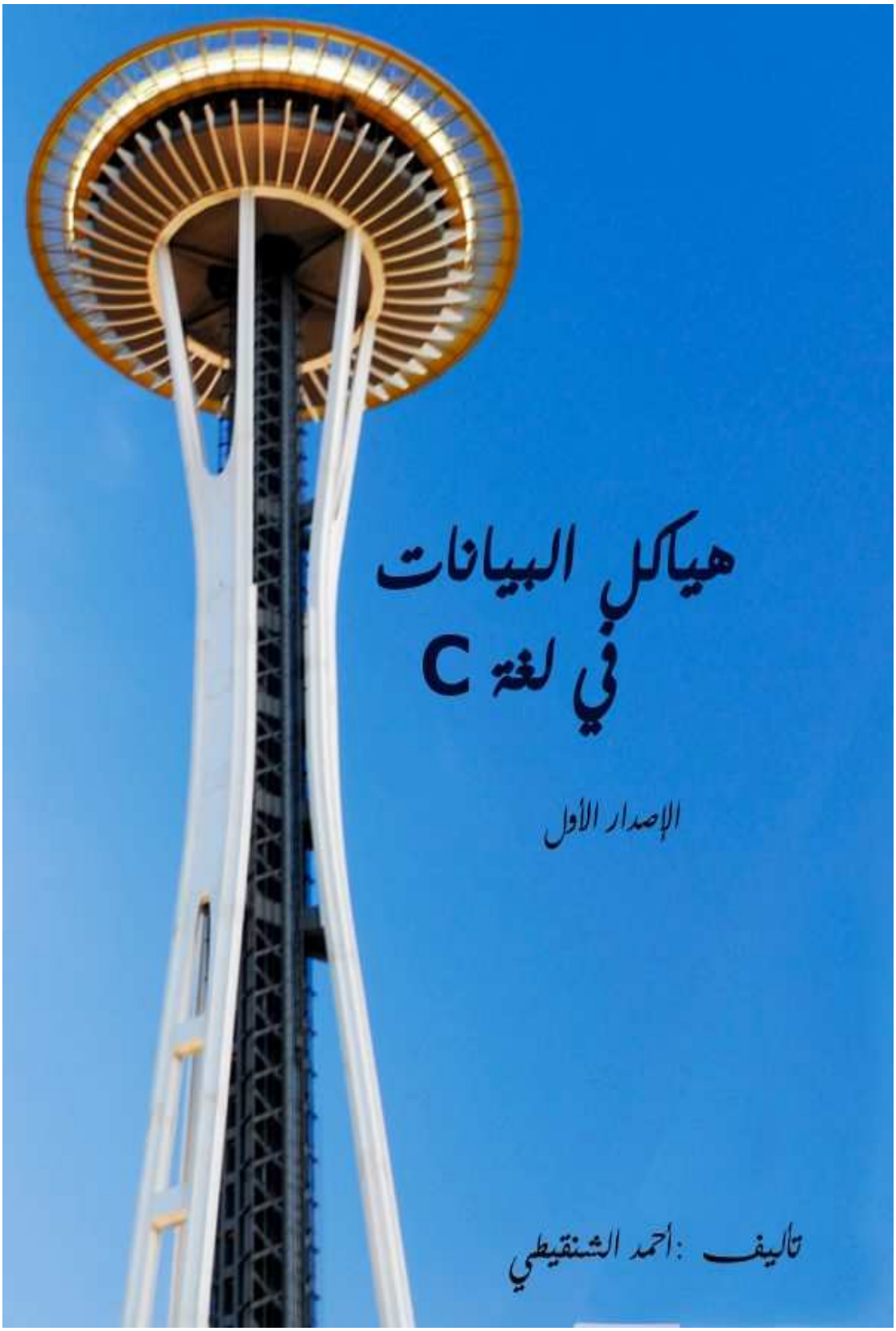

هياكل البيانات
في لغة C
الإصدار الأول
تاليف :أحمد الشنقيطي

الحمد لله الذي بحمده يُستفتح كل كتاب و بذكره يُصدر كل خطاب وبفضله يتنعم أهل النعيم في دار الجزاء و الثواب والصلاة و السلام على سيد المرسلين و إمام المتقين المبعوث رحمة للعالمين محمد بن عبد الله الصادق الأمين و على صحابته الأخيار و من تبعهم بإحسان إلى يوم الدين أما بعد :

تم بحمد الله الانتهاء من الإصدار الأول من سلسلة هياكل البيانات في لغة C. قمتُ بتقسيم الكتاب إلى جزئين, كل جزء يحتوي على شرح 4 أنواع من هياكل البيانات, هذا الإصدار يشرح الهياكل التالية:

- Singly Linked List
- Doubly Linked List
- Stacks
- Queues

و الإصدار القادم إن شاء الله سأركز فيه على الهياكل التالية :

- Trees
- Binary Trees
- Hash tables
- Graphs

الكاتب في سطور:

الاسم: أحمد بن محمد

اللقب: الشنقيطي

سنة الميلاد: 1992

الدولة: بلاد شنقيط و أرض المليون شاعر .. موريتانيا

الهواية: programming & Security

المستوى الأكاديمي: خريج كلية العلوم و التقنيات.

للتواصل: ahmed.ould_mohamed@yahoo.fr

كُتب بتاريخ 02/02/2013

لمن هذه الدروس ؟

هذه الدروس مُوجهة إلى كل من لديه معرفة بأساسيات السي مثل المصفوفات, التراكيب و المؤشرات و يتطلع إلى دراسة هياكل البيانات المتقدمة في لغة السي.

الجزء الأول – القوائم المتصلة البسيطة

- **الجانب النظري**
  - تعريف
  - نبذة تاريخية
  - قالوا عن الـ Linked List
  - أيهما أفضل, المصفوفة الديناميكية أم القائمة المتصلة ؟
  - مفهوم القوائم المتصلة
- **الجانب التطبيقي** (العمليات الأكثر استخداما في القوائم)
  - مدخل
  - تهيئة القائمة (إنشاء أول عقدة)
  - إضافة عقدة جديدة.
  - حذف عقدة معينة.
  - حساب طول القائمة.
  - دمج قائمتين في قائمة واحدة
  - حذف قائمة
- **اختبر قدرتك**

الجانب النظري

تعريف

القائمة المتصلة عبارة عن مجموعة من العُقد, مُخزنة في الذاكرة بشكل متصل و غير متسلسل و هذا أحد أبرز أوجه الخلاف بين القوائم المتصلة و المصفوفات.

نبذة تاريخية

كانت تُعرف القوائم المتصلة باسم NSS memory و تم تصميمها خلال السنتين 1955-1956, من طرف الثلاثي Allen Newell, Cliff Shaw و Herbert Simon برعاية المؤسسة الأمريكية للبحوث RAND Corporation.

كانت القوائم المترابطة هي البُنية الأساسية في لغتهم Information Processing Language (IPL) و كان المخترعون الثلاثة يستخدمون IPL لتطوير مجموعة من برامج الذكاء الاصطناعي, مثل Logic Theory Machine, General Problem Solver بالإضافة إلى Chess (لعبة الشطرنج).

نُشرت أعمال الفريق حول القوائم المتصلة في الـ IRE Transactions on Information Theory في سنة 1956 و عُقدت العديد من المؤتمرات خلال الفترة [1] 1957 – 1959. أما التمثيل الحالي للقوائم المتصلة (حيث تتكون القائمة من مجموعة عقد مُرتبطة فيما بينها بواسطة أسهم) فقد تم تنشره في شهر فبراير من عام 1957 تحت عنوان Programming the Logic Theory Machine [2].

في عام 1975 حصل الثنائي Allen Newell و Herbert Simon على جائزة Turing لمساهمتهم الفعالة في علم الذكاء الاصطناعي و التعامل مع القوائم.

---

(1) Proceedings of the Western Joint Computer Conference en 1957 et 1958 et Information Processing en 1959 (première réunion de l'International Conference on Information Processing de l'UNESCO)

(2) Programming the Logic Theory Machine de Allen Newell et Cliff Shaw, Proceedings of the 1957 Western Joint Computer Conference, février 1957.

قالوا عن الـ Linked List

بطبيعة الحال, يُمكن تعريف القوائم المتصلة بأكثر من طريقة لذا اقتطفت لكم بعض التعريفات التي وردت في أهم الكتب المتعلقة بهياكل البيانات:

- يقول Seymour Lipschutz في كتابه The data structures (Courses and problems) : "القائمة عبارة عن مجموعة خطية من عناصر البيانات".
- بينما يرى John Rast Hubbard في كتابه Java Data Structures أن "القائمة عبارة عن حاوية متسلسلة العناصر و قادرة على إدراج و إزالة العناصر بشكل مطرد محليا, بمعنى : بغض النظر عن حجم الحاوية".
- أما Alain-Bernard Fontaine فيرى في كتابه The C++ Standard Template Library أن "القوائم عبارة عن حاويات مُخصصة للقيام بعمليات معينة (مثل الإدراج و الإزالة) حيث تتم هذه العمليات في وقت ثابت مهما كان موقع العنصر داخل الحاوية".
- و يعتبر الثلاثي Thomas Cormen, Charles Leiserson و Ronald Rivest في كتابهم Introduction to Algorithms أن "القائمة المتصلة عبارة عن هيكل بيانات يتم فيه ترتيب الكائنات بشكل خطي ولكن بخلاف المصفوفات التي تُحدد فيها العناصر عن طريق ترقيم الخانات, يتم تحديد عناصر القائمة المتصلة عن طريق مؤشر في كل كائن"

أيهما أفضل, المصفوفة الديناميكية أم القائمة المتصلة ؟

عادة ما يكون السؤال الذي يطرح نفسه عند المقارنة بين هياكل البيانات, هو:

ما مدى سهولة الوصول إلى عناصر المجموعة ؟ و كذا التنقل بين مختلف العناصر و إجراء العمليات الأكثر استعمالا ؟

للإجابة على هذا السؤال قُمنا بحساب التعقيد الزمني للعمليات الأكثر استعمالا:

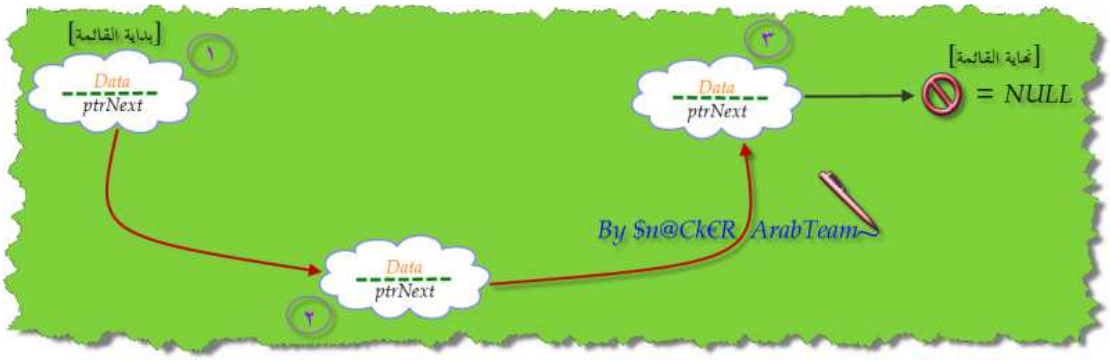
العملية	المصفوفة	القائمة المتصلة
الإضافة	$O(n)$	$O(1)$
الحذف	$O(n)$	$O(1)$
البحث في مجموعة غير مرتبة	$O(n)$	$O(n)$
البحث في مجموعة مرتبة	$O(log_2 n)$	$O(n)$
المرور على كافة عناصر المجموعة	$O(n)$	$O(n)$

لإضافة عنصر معين في الخانة رقم i من المصفوفة X يجب إزاحة size-i عنصر حيث size هو طول المصفوفة و بالتالي عملية الإضافة في المصفوفات تستغرق وقتاً لا يُستهان بها و يزداد الأمر سوءا كلما كثُرت العناصر. نفس الشيء يحدث مع عملية الحذف.

أما في القوائم المتصلة فعملية الحذف أو الإضافة تأخذ نفس الوقت بغض النظر عن طول القائمة أو المكان المُراد إضافة (حذف) العُقدة فيه (منه).

نلاحظ أن القوائم المتصلة تكون أسرع في عمليتي الإضافة و الحذف بينما تكون المصفوفة المُرتبة أسرع في عملية البحث و يتساوى الاثنان عند المرور على جميع العناصر.

مفهوم القوائم المتصلة



ذكرنا آنفا أن طريقة تخزين عناصر المصفوفة تختلف عن طريقة تخزين عناصر القائمة إذْ أن عناصر الأولى تُخزَّن في أماكن متتابعة في الذاكرة أما القوائم فلا يُشترط تتابع عناصرها لأن الوصول إلى أي عُقدة من القائمة يتم عن طريق مؤشر في العُقدة السابقة أما المصفوفات فيتم الوصول إلى خاناتها عن طريق الترقيم لذا لزمها تتابع الخانات.

القوائم المتصلة تنتمي إلى عائلة الـ Dynamic Data Structures لذا تجد أن عملية حجز الذاكرة تتم أثناء عمل البرنامج باستخدام malloc مثلا.

قبل أن ننتقل إلى الجانب التطبيقي, تذكر النقاط التالية جيدا لأنك ستحتاجها لفهم العمليات الأكثر استخداما في القوائم :

- ✓ تتكون القائمة المتصلة من مجموعة عُقد ترتبط فيما بينها عن طريق المؤشرات.
- ✓ كل عُقدة تحتوي على جزئين, الجزء الأول يحتوي على بيانات العقدة و الجزء الثاني عبارة عن مؤشر يُشير إلى العقدة الموالية.
- ✓ للوصول إلى عُقدة معينة, يجب الذهاب دائماً من أول عُقدة. و يتم الانتقال من العقدة الموالية عن طريق المؤشر الموجود في العُقدة السابقة.
- ✓ دائما ما يُشير المؤشر الموجود في آخر عقدة إلى NULL.
- ✓ عندما تكون أول عقدة هي آخر عُقدة, نقول أن القائمة فارغة.
- ✓ في القوائم المتصلة البسيطة (Singly Linked List) تتم الحركة في اتجاه واحد, انطلاقا من العُقدة الأولى باتجاه العقدة الأخيرة.
- ✓ إذا كنتَ تريد التحرك في كلا الاتجاهين, يُمكنك استخدام القوائم المتصلة المُضاعفة (Doubly Linked List).

الجانب التطبيقي

مدخل

في بقية هذه الفقرة, سنعتبر أن القائمة التي نعمل عليها تحتوي على بيانات مجموعة من الموظفين في شبكة محلية تابعة لشركة مسابقات, كل مُوظف يملك رقم دخول و بريد الكتروني, نفترض أن رقم الدخول يُميز كل موظف عن الآخر بمعنى أنه لا يُمكن أن نجد مُوظفيْن يملكان نفس رقم الدخول. كما أن أرقام الدخول يجب أن تكون أكبر تماماً من الصفر.

قامت الشركة بتنظيم المسابقة التالية لموظفيها على النحول التالي:

كل مُوظف سيختار حرف عشوائي من A إلى Z. يفوز الموظف إذا تطابق الحرف المُختار مع أول حرف من مُعرَّفه (الجزء الموجود قبل @ من البريد الالكتروني) و يخسر في الحالة المعاكسة.

الإعلان عن القائمة التي ستضم بيانات الموظفين سيكون هكذا:

```
struct singlyLinkedList {
    int login;
    char randomCharacter;
    char email[45];
    singlyLinkedList * ptrNext;
};
```

كلما في الأمر أننا قمنا بالإعلان عن بنية باسم singlyLinkedList تحتوي على 4 عناصر, العناصر الثلاثة الأولى تُمثل بيانات الموظف و العنصر الرابع عبارة عن المؤشر الذي يُشير إلى الموظف الموالي.

الخطوة التالية تتمثل في بناء القائمة عن طريق الإعلان عن المؤشر الذي سيُشير لاحقا إلى أول عُقدة, قُمنا بإعطاء اسم مستعار لمؤشر القائمة من أجل تسهيل و تنظيم الكتابة:

```
typedef struct singlyLinkedList* list;
```

تهيئة القائمة (إنشاء أول عقدة)

نأتي الآن إلى كيفية تهيئة قائمة الموظفين من خلال إنشاء حساب لموظف جديد و إسناد قيم ابتدائية لكافة البيانات:

```
bool init(list &sll) {
    sll = (list) malloc(sizeof (singlyLinkedList));
    if (sll == NULL) return false;
    else {
        sll->login = 0;
        sll->randomCharacter = ' ';
        memset(sll->email, 0, sizeof (sll->email));
        sll->ptrNext = NULL;
        return true;
    }
}
```

ملاحظات هامة:

- إذا كانت P عبارة عن بنية تحوي (مثلاً) عنصرين x و y فإن الوصول إلى عناصر البنية يكون هكذا P.x Or P.y.
- إذا كان Q مؤشر لبنية من نوع P, فإن الوصول إلى عناصر البنية يكون هكذا : Q->x Or Q->y.
- لغة C لا تدعم التمرير بالمرجع (Pass by reference) على عكس C++.
- تستخدم C تمرير المؤشرات الثابتة بدلا من تمرير المراجع لكنني أفضل دائما التمرير بالمرجع و لحسن الحظ, معظم مترجمات C الحالية (المُحتكة بـ C++) تدعمه.

بالنسبة للدالة init فتستقبل وسيط واحد عبارة عن مرجع (Reference) لقائمة الموظفين, في السطر الأول قمنا بحجز ذاكرة للمؤشر الذي سيُشير إلى أول عقدة في القائمة, إذا فشلت عملية الحجز ستعيد الدالة false و ينتهي الأمر, أما إذا مرت عملية الحجز بسلام فهذا يدل على أن المؤشر sll أصبح يُشير إلى منطقة من الذاكرة تحوي 4 عناصر (رقم الدخول, الحرف العشوائي, البريد الالكتروني و المؤشر) و في هذه الحالة سنقوم بإسناد قيمة ابتدائية لكل عنصر على النحو التالي:

إسناد القيمة 0 إلى المتغير Login و تخزين المسافة البيضاء داخل المتغير randomCharacter و إسناد المؤشر NULL إلى ptrNext أما المتغير email فله حالته الخاصة, لأنه عبارة عن نوع مُركب و ليس نوع بسيط مثل (int, float, char, bool, ..) الأنواع المُركبة (مثل المصفوفات و التراكيب ,..) يتم التعامل معها بصفة مختلفة لذا قُمنا باستدعاء الدالة memset لتصفير الذاكرة التي يُشير إليها المؤشر email, الدالة memset تستقبل 3 معاملات, المعامل الأول هو المؤشر المُراد تصفير ذاكرته و المعامل الثاني هو القيمة المراد وضعها داخل المنطقة التي يشير إليها المؤشر و المعامل الثالث هو عدد البايتات أو كمية البيانات المُؤشر عليها.

بعد تهيئة العناصر الأربعة تقوم الدالة بإعادة true كإشارة إلى نجاح العملية.

نأتي الآن إلى تضمين المكتبات اللازمة بالإضافة إلى شرح مختصر لمحتوى الدالة الرئيسية :

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

int main() {
    list mySimpleList = NULL;
    if (!init(mySimpleList)) {
        printf("Insufficient Memory\n");
    } else {
        printf("Login : %d\nRandom Character : %c\nE-mail : %s\n",
                mySimpleList->login, mySimpleList->randomCharacter, mySimpleList->email);
    }
    return 0;
}
```

الدالة printf موجودة في المكتبة stdio.h و الماكرو NULL موجود في المكتبة stdlib.h و الدالة memset موجودة في المكتبة string.h لذا قمنا باستدعاء المكتبات الثلاثة معاً.

في بداية الدالة الرئيسية قمنا بالإعلان عن قائمة جديدة و أسندنا لها العنوان NULL وهذه الخطوة ضرورية جدا في تهيئة المؤشرات أيا كانت.

بعد ذلك, قمنا بتمرير القائمة mySimpleList إلى الدالة init كوسيط ثم تحققنا من القيمة المُعادة من طرف الدالة, إذا كانت false سيتم إظهار رسالة تنبيه على الشاشة و إلا فسيتم إظهار البيانات الابتدائية للموظف.

و هذه صورة لمُخرجات الكود :

```
Login : 0
Random Character :
E-mail :

Process returned 0 (0x0)   execution time : 0.405 s
Press any key to continue.
```

إضافة عقدة جديدة

لإضافة عقدة جديدة إلى القائمة, يجب أن نمر بالخطوات التالية:

- حجز مساحة من الذاكرة للعقدة الجديدة.
- تهيئة كافة عناصر العقدة بما في ذلك المؤشر.
- إضافة العقدة و تحديث القائمة.

و بالتالي, دالة الإضافة ستكون هكذا:

```
const char* aplphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";

bool addToTheTopOfTheList(list &sll, int l, char em[45]) {
    list newNode;
    if ((newNode = (list) malloc(sizeof (singlyLinkedList))) == NULL)
        return false;
    else {
        newNode->login = l;
        newNode->randomCharacter = aplphabet[rand() % 26];
        strcpy(newNode->email, em);
        newNode->ptrNext = sll;
        sll = newNode;
        return true;
    }
}
```

كالعادة, إذا لم تنجح عملية الحجز ستُعيد الدالة false و إلا فالقيمة المُعادة ستكون true, هذا من جهة. من جهة أخرى, تستقبل الدالة 3 وسائط, الأول عبارة عن مرجع (Reference) لقائمة الموظفين و الوسيط الثاني عبارة عن رقم دخول الموظف الجديد و الوسيط الثالث عبارة عن البريد الالكتروني. بطبيعة الحال, من غير المنطقي أن نضع الحرف العشوائي ضمن وسائط الدالة لأن قيمته تعتمد على توليد عشوائي.

في البداية, أسندنا قيمة الوسيط L إلى المتغير login ثم قمنا باختيار حرف عشوائي من المصفوفة aplphabet و أسندناه إلى المتغير randomCharacter. بعد ذلك, استخدمنا الدالة strcpy لنسخ محتوى الوسيط em داخل المصفوفة email ثم جعلنا المؤشر الحالي يُشير إلى أول عُقدة في القائمة ثم قمنا بتحديث القائمة من خلال جعل العُقدة الجديدة هي الأولى. نأتي الآن إلى تضمين المكتبات اللازمة بالإضافة إلى تعليق موجز حول الـ main :

```
#include <time.h>

int main() {
    list mySimpleList = NULL;
    char mail[45] = "Sn@CkeR@ArabTeam.com";
    init(mySimpleList);
    srand(time(NULL));
    if (addToTheTopOfTheList(mySimpleList, 19, mail)) {
        fprintf(stderr, "Insufficient Memory\n");
        return EXIT_FAILURE;
    } else {
        display(mySimpleList);
        return EXIT_SUCCESS;
    }
}
```

تنويه:

قمتُ بتقسيم الكود الكامل إلى مجموعة فقرات للتوضيح, فمثلا لم أقم بتضمين المكتبات التي قمتُ بتضمينها سابقا للاختصار, لذا تجد أن كل جزء من الكود يكون مُرتبطاً بالجزء السابق.

في البداية, قمنا بتضمين المكتبة time.h لوجود كلا من rand و srand. ثم قمنا بالإعلان عن بريد الكتروني باسم mail و استدعينا دالة الإضافة ثم تحققنا من القيمة المُعادة كما فعلنا سابقا, إذا تمت إعادة false نُظهر رسالة الخطأ و إلا فسنستدعي الدالة display المسئولة عن إظهار بيانات المُوظفين (سنشرح هذه الدالة لاحقاً).

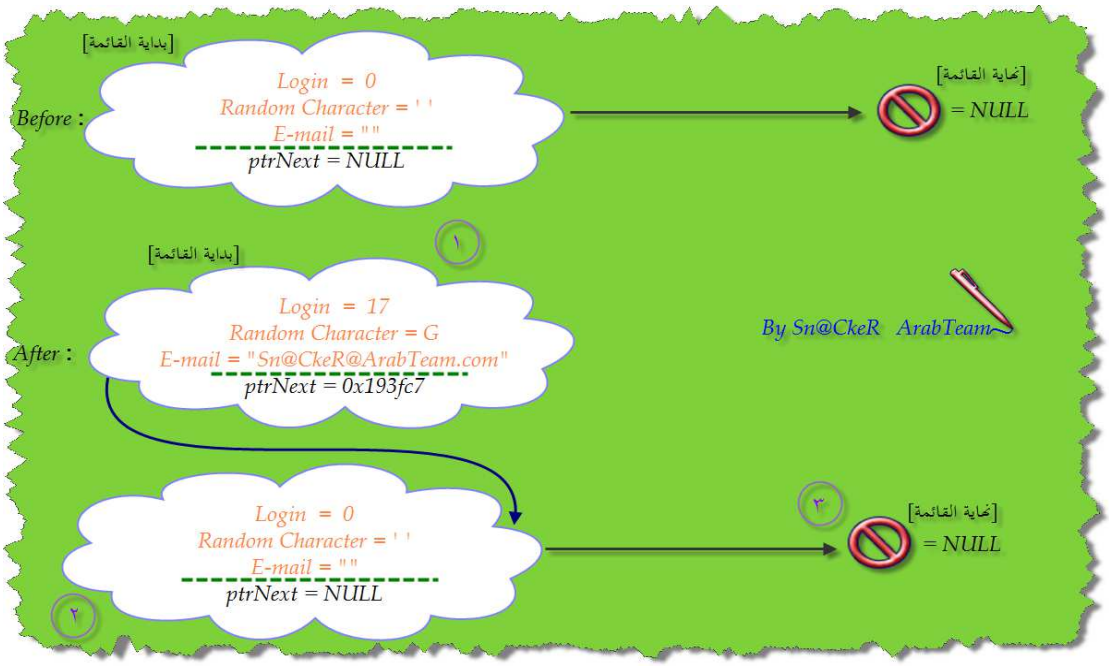
مُخرجات الكود ستكون على هذا النحو:

```
Login : 19
Random Character : K
E-mail : Sn@CkeR@ArabTeam.com
--------------------
Login : 0
Random Character :
E-mail :
--------------------

Process returned 0 (0x0)   execution time : 1.575 s
Press any key to continue.
```

لاحظ أنه تم إدراج العُقدة الجديدة قبل العُقدة الأولى و هذا ما يُسمى بالإضافة في بداية القائمة, يُمكننا أيضا أن نُضيف العُقدة في نهاية القائمة أو في أي مكان آخر و هنا تكمن أحد أبرز نقاط القوة لدى القوائم و هي المرونة.

الصورة التالية تُوضح مفهوم الإضافة في بداية القائمة:



بالنسبة لإضافة العُقدة في نهاية القائمة فستكون هكذا:

```
bool addAtTheEndOfTheList(list &sll, int l, char em[45]) {
    list newNode, tempNode = sll;
    if ((newNode = (list) malloc(sizeof (singlyLinkedList))) == NULL)
        return false;
    else {
        newNode->login = l;
        newNode->randomCharacter = aplphabet[rand() % 26];
        strcpy(newNode->email, em);
        newNode->ptrNext = NULL;
        while (tempNode->ptrNext != NULL) {
            tempNode = tempNode->ptrNext;
        }
        tempNode->ptrNext = newNode;
        return true;
    }
}
```

لا شيء جديد, سوى إضافة الحلقة while من أجل المرور على كافة العقد وصولا إلى العقدة الأخيرة ثم ربط العقدة الجديدة بنهاية القائمة. بشكل عام, الخطوات المُتبعة عند إضافة عُقدة في نهاية القائمة تكون كالآتي:

- ♣ حجز مساحة من الذاكرة للعقدة الجديدة.
- ♣ تهيئة كافة عناصر العقدة بما في ذلك المؤشر.
- ♣ الانتقال إلى آخر عُقدة من القائمة.
- ♣ ربط آخر عُقدة بالعقدة الجديدة.

يُمكننا أيضا إضافة عقدة جديدة في أي مكان من القائمة, لنفترض مثلا أننا نريد إدراج مُوظف جديد بعد موظف آخر يتم تحديد رقمه. في هذه الحالة ستكون الخُطوات المُتبعة هي:

- ♣ البحث عن العُقدة المُرافقة لرقم الموظف, توجد حالتان:
  - ❖ إذا وصلنا إلى NULL فهذا يعني أن رقم الموظف غير موجود, حينها نقوم بإعادة false.
  - ❖ إذا وصلنا إلى العُقدة المطلوبة نقوم بالآتي:
- ♣ حجز مساحة من الذاكرة للعقدة الجديدة.
- ♣ تهيئة كافة عناصر العقدة بما في ذلك المؤشر.
- ♣ تخزين عنوان العقدة الموالية في متغير مؤقت.
- ♣ جعل العُقدة الحالية تُشير إلى العُقدة الجديدة.
- ♣ ربط عنوان العُقدة الجديدة بالمتغير المؤقت.

و بالتالي دالة الإضافة ستكون هكذا:

```
bool addAfterACertainNode(list &sll, int l, char em[45], int num) {
    list newNode, tempNode = sll, afterNode;
    while (tempNode->ptrNext != NULL && tempNode->login != num) {
        tempNode = tempNode->ptrNext;
    }
    if (tempNode->login == num) {
        if ((newNode = (list) malloc(sizeof (singlyLinkedList))) == NULL)
            return false;
        newNode->login = l;
        newNode->randomCharacter = aplphabet[rand() % 26];
        strcpy(newNode->email, em);
        afterNode = tempNode->ptrNext;
        tempNode->ptrNext = newNode;
        newNode->ptrNext = afterNode;
        return true;
    } else return false;
}
```

إذا لم تنجح عملية الحجز أو لم يتم العثور على رقم الموظف ستُعيد الدالة false و إلا فستعيد الدالة true بعد أن تتم إضافة العُقدة الجديدة و تحديث القائمة.

الآن أصبح من السهل جداً كتابة دالة تقوم بتعديل بيانات مُوظف مُعين, فقط بدلا من إدراج عُقدة جديدة, نقوم بتغيير بيانات العُقدة التي توقفت عندها الحلقة while ما لم تكن تلك العقدة فارغة.

نأتي الآن إلى شرح كيفية عمل دالة الإظهار التي تقوم بعرض محتويات قائمة المُوظفين:

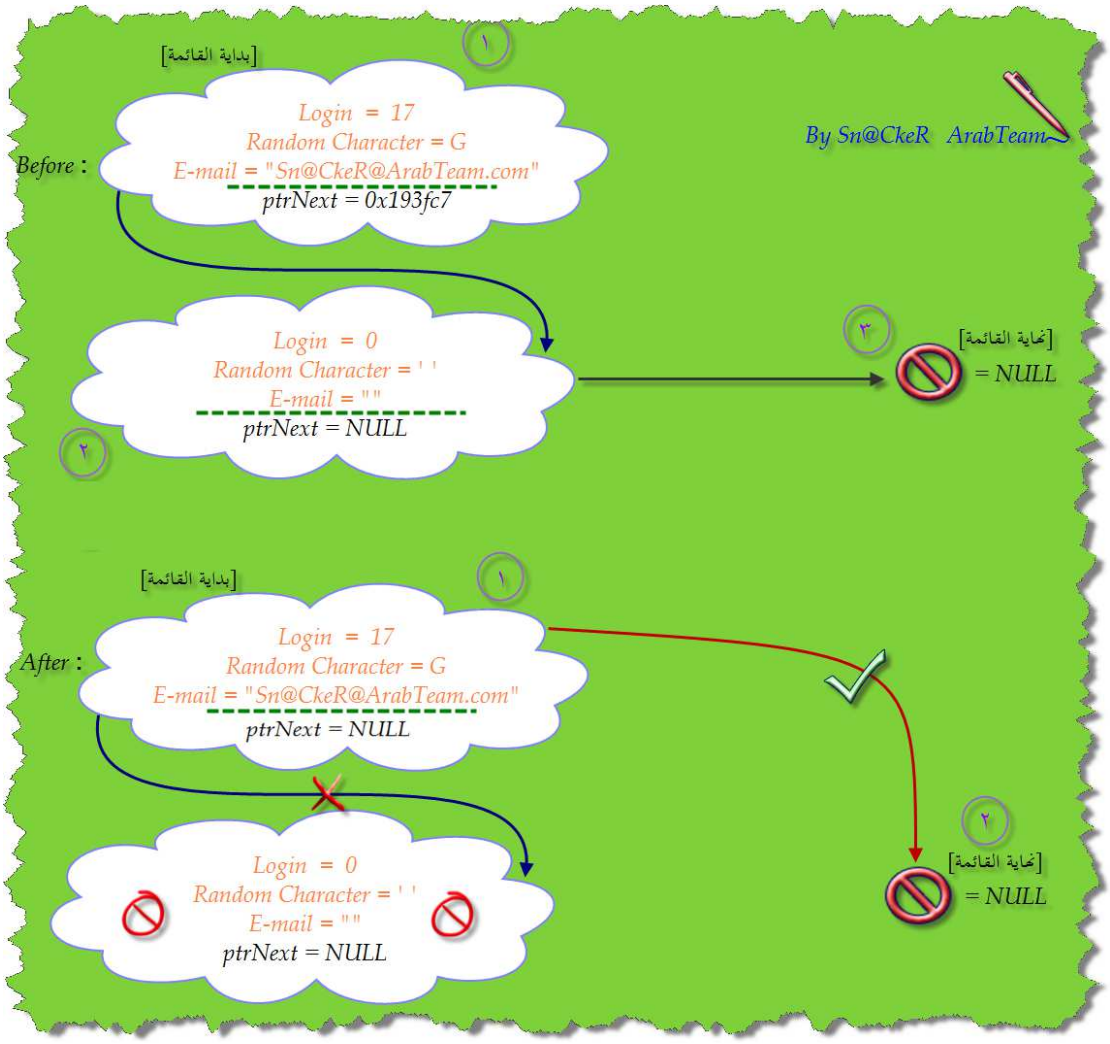
```
void viewList(list &sll) {
    list tempNode;
    tempNode = sll;
    while (tempNode != NULL) {
        printf("Login : %d\nRandom Character : %c\nE-mail : %s\n",
                tempNode->login, tempNode->randomCharacter, tempNode->email);
        puts("--------------------");
        tempNode = tempNode->ptrNext;
    }
}
```

الفكرة بسيطة جداً و هي كالتالي:

حتى لا نفقد بيانات المُوظفين نقوم بإنشاء نُسخة من القائمة المُرسلة كوسيط, ما دامت العُقدة الحالية غير فارغة نقوم بالتقدم خطوة إلى الأمام بعد أن نُظهر كافة بيانات العُقدة. فقط هذا كل شيء :-)

يُمكننا حساب طول القائمة بنفس الفكرة, فقط بدلا من إظهار البيانات نقوم بزيادة العداد ثم نُعيد قيمة العداد عند الوصول إلى آخر عُقدة. توجد أيضا طريقة أخرى لإظهار محتوى القائمة باستخدام التراجع, سنشرحها لاحقاً.

حذف عقدة مُعينة



كما فعلنا سابقا مع عملية الإضافة, يمكننا أيضا حذف عُقدة من البداية, النهاية أو أي مكان آخر من القائمة.

فمثلا, الدالة التالية تقوم بحذف آخر عُقدة من القائمة :

```
void removeTheLast(list &q) {
    list tempNode, delNode;
    tempNode = q;
    while ((tempNode->ptrNext)->ptrNext != NULL) {
        tempNode = tempNode->ptrNext;
    }
    delNode = tempNode->ptrNext;
    tempNode->ptrNext = NULL;
    free(delNode);
}
```

الخُطوات المُتبعة هي:

- إنشاء نُسخة من القائمة حتى لا نفقد بيانات الموظفين.
- الانتقال إلى العُقدة القبل الأخيرة (نتقدم بخطوة واحدة إذا كانت العقدة الموالية للعقدة الموالية غير فارغة).
- تخزين عنوان العقدة الأخيرة في متغير مؤقت وجعل العُقدة القبل الأخيرة تُشير إلى NULL.
- تحرير العنوان الذي كان يُشير إلى العُقدة الأخيرة.

أما إذا أردنا حذف عقدة من بداية القائمة فستكون الخطوات كما يلي:

- تخزين عنوان العقدة الأولى داخل متغير مؤقت.
- جعل مؤشر القائمة يُشير إلى العقدة الثانية.
- تحرير المؤشر الذي يُشير إلى العقدة الأولى.

و بالتالي دالة الحذف في هذه الحالة ستكون هكذا:

```
void removeTheFirst(list &sll) {
    if (sll != NULL) {
        list delNode = sll;
        sll = sll->ptrNext;
        free(delNode);
    }
}
```

لاحظ أنه تم التأكد أن القائمة غير فارغة قبل إجراء عملية الحذف و هذا مُهم جدا إذْ يجب التأكد من مثل هذه الحالات قبل القدوم على تنفيذ العملية.

لم أتحقق من أشياء كهذه في الدوال السابقة نظراً لأنني افترضتُ أن استدعاء الدوال يجب أن يكون بشكل تتابعي, في هذه الحالة لن نحصل على أي خطأ. أيضاً لم أُرد إرباك القارئ من خلال مُعالجة عدة أخطاء في نفس الوقت.

نأتي الآن إلى كيفية حذف عُقدة عن طريقة تحديد رقم المُوظف:

```
bool removeAfterACertainNode(list &sll, int num) {
    list tempNode, delNode;
    if (sll->login == num) {
        removeTheFirst(sll);
        return true;
    }
    tempNode = sll;
    while ((tempNode->ptrNext)->ptrNext != NULL && (tempNode->ptrNext)->login != num) {
        tempNode = tempNode->ptrNext;
    }
    if ((tempNode->ptrNext)->login != num)
        return false;
    else {
        delNode = tempNode->ptrNext;
        tempNode->ptrNext = delNode->ptrNext;
        free(delNode);
        return true;
    }
}
```

إذا كان رقم الموظف موجود في العُقدة الأولى نقوم باستدعاء الدالة removeTheFirst لأن الانتقال في while يكون بمقدار خطوتين إلى الأمام و بالتالي سيتم تجاوز العُقدة الأولى, هذا من جهة.

من جهة أخرى, عند الخروج من while سنكون أمام خيارين, الخيار الأول هو عدم وجود رقم المُوظف حينها سنقوم بإعادة false و تنتهي المهمة, إذا تجاوزنا الـ if بسلام فهذا يعني أن رقم المُوظف موجود داخل القائمة, لذا سنقوم بتخزين العقدة التي يوجد فيها الرقم في المتغير delNode ثم نُحرر عنوان تلك العُقدة بعد أن نتقدم خطوة واحدة إلى الأمام.

حساب طول القائمة

الفكرة بسيطة جداً وهي كالآتي:

إذا وصلنا إلى العقدة الأخيرة نقوم بإعادة الصفر و إلا فهذا يعني أنه توجد عقدة أخرى (و هي العُقدة الحالية) بالإضافة إلى العُقد الموجودة في بقية القائمة (إن وُجدت).

و بالتالي دالة حساب الطول ستكون هكذا:

```
int lengthOfTheList(list sll) {
    return (sll == NULL ? 0 : lengthOfTheList(sll->ptrNext) + 1);
}
```

قبل أن ننتقل إلى الفقرة الموالية, ستقوم بكتابة الدالة التي من خلالها نستطيع معرفة فوز أو خسارة مُوظف مُعين:

```
bool isWin(list &sll) {
    return (*sll->email == sll->randomCharacter);
}
```

ينجح المُوظف إذا تساوي أول حرف من بريده الالكتروني مع الحرف العشوائي الخاص به و يخسر في الحالة المُعاكسة.

دمج قائمتين في قائمة واحدة

دالة الدمج تستقبل وسيطين يُمثلان القائمتين المُراد دمجهما :

```
list mergeTwoLists(list firstList, list secondList) {
    if (!firstList)
        return secondList;
    list tmp = firstList;
    while (tmp->ptrNext)
        tmp = tmp->ptrNext;
    tmp->ptrNext = secondList;
    return firstList;
}
```

إذا كانت القائمة الأولى فارغة, فهذا يعني أن دمج القائمتين يُعطي القائمة الثانية لذا تمت إعادتها. في الحالة المُعاكسة نقوم بالانتقال إلى آخر عُقدة من القائمة الأولى ثم نربطها بأول عُقدة من القائمة الثانية.

حذف القائمة

دالة الحذف فكرتها كالآتي : ما دامت القائمة غير فارغة, نقوم بحذف العقدة الأولى. فقط ! :-)

```
void clearTheList(list &sll) {
    int length = lengthOfTheList(sll);
    while (length-- > 0)
        removeTheFirst(sll);
}
```

اختبر قدراتك

قررت الشركة كتابة برنامج صغير يُساعدها في تسيير المُوظفين على مرحلتين:

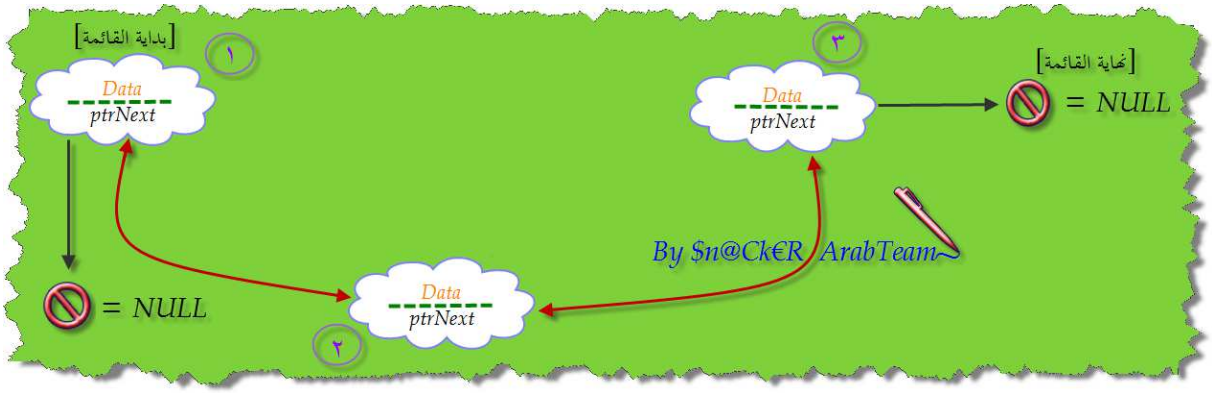
1. ترتيب المُوظفين تصاعدياً حسب أرقام الدخول.
2. تقسيم قائمة المُوظفين إلى قائمتين, الأولى تحتوي على كافة المُوظفين الذين فازوا في المسابقة و القائمة الثانية تحتوي على بقية المُوظفين.

قم بكتابة دالة خاصة بكل مرحلة ثم قم باختبار الدوال في برنامج رئيسي موجود في ملف مستقل.

الجزء الثاني – القوائم المتصلة المزدوجة

تعريف

تتميز القوائم المزدوجة بوجود مؤشرين في كل عُقدة, المؤشر الأولى يُشير إلى العُقدة السابقة و المؤشر الثاني يُشير إلى العقدة الموالية.



بشكل عام, المؤشرات هي ما يميز طبيعة الحركة في القوائم, أياً كان نوعها, و في حالتنا هذه فإن القوائم المزدوجة تتميز بالقدرة على الحركة في كلا الاتجاهين نظرا لوجود مؤشر سابق و آخر لاحق في كل عقدة.

تنويه:

في بقية المقالة :

- سأقوم بتقسيم الكود الكامل إلى مجموعة دوال للتوضيح.
- إذا اجتمعت في الكود عدة دوال تشترك في نوعية الخطأ (فشل عملية الحجز مثلاً) سيتم التحقق من دالة واحدة فقط و ذلك تجنباً للتكرار.
- كل جزء من الكود سيكون مُرتبطاً بالجزء السابق له.

الإعلان عن القائمة

عند التعامل مع القوائم, يُستحسن دائما استخدام قائمة مُساعدة تأتي فائدتها في تخزين بعض المعلومات التي تخص القائمة الأصلية, مثل عناوين العُقد الرئيسية (الأولى و الأخيرة مثلاً) و طول القائمة و ما إلى ذلك ..

لتبسيط الأمور, نفترض أن القائمة التي سنعمل عليها تحوي عنصرا واحدا عبارة عن متغير من نوع int. الإعلان عن القائمة سيكون هكذا:

```
typedef struct doublyLinkedList {
    int elem;
    struct doublyLinkedList *ptrNext;
    struct doublyLinkedList *ptrPrev;
};

typedef doublyLinkedList* list;
```

لا شيء جديد, فقط قمنا بالإعلان عن قائمة تحوي 3 عناصر : متغير من نوع int ومؤشر على العقدة السابقة و آخر على العقدة اللاحقة. ثم قمنا بإعطاء اسم مستعار لمؤشر القائمة.

بالنسبة للقائمة المُساعدة فستكون هكذا:

```
typedef struct aboutList {
    int lengthOfTheList;
    list first;
    list last;
};

typedef aboutList* dbll;
```

القائمة المُساعدة تحتوي على متغير من نوع int يُمثل طول القائمة و مؤشرين, الأول يُشير إلى بداية القائمة الأصلية و الثاني يُشير إلى نهايتها.

تهيئة القائمة (إنشاء أول عقدة)

نأتي الآن إلى كيفية تهيئة القائمة من خلال إنشاء عقدة جديدة و إسناد قيم ابتدائية لكافة البيانات:

```
bool init(dbll &argDbll) {
    argDbll = (dbll) malloc(sizeof (doublyLinkedList));
    if (argDbll == NULL) return false;
    else {
        argDbll->lengthOfTheList = 0;
        argDbll->first = NULL;
        argDbll->last = NULL;
        return true;
    }
}
```

الدالة init تستقبل وسيط واحد عبارة عن مرجع (Reference) للقائمة المزدوجة, في السطر الأول قمنا بحجز ذاكرة للمؤشر الذي سيُشير إلى أول عقدة في القائمة, إذا فشلت عملية الحجز ستعيد الدالة false و ينتهي الأمر, أما إذا مرت عملية الحجز بسلام فهذا يدل على أن المؤشر argDbll أصبح يُشير إلى منطقة من الذاكرة تحوي 3 عناصر (المتغير الصحيح, المؤشر السابق و المؤشر اللاحق) و في هذه الحالة سنقوم بإسناد قيمة ابتدائية لكل عنصر.

بعد تهيئة العناصر الثلاثة تقوم الدالة بإعادة true كإشارة إلى نجاح العملية.

ملاحظات هامة:

- في حالة عدم وجود العُقدة السابقة يُشير المؤشر السابق إلى NULL.
- في حالة عدم وجود العُقدة الموالية يُشير المؤشر اللاحق إلى NULL.
- إذا كانت القائمة تحتوي على عُقدة واحدة فهذا يعني دمج الملاحظتين السابقتين.
- العمليات المختلفة (تهيئة, إضافة, حذف, ..) سيتم تطبيقها على القائمة الأصلية من خلال القائمة المُساعدة.

نأتي الآن إلى تضمين المكتبات اللازمة بالإضافة إلى شرح مختصر لمحتوى الدالة الرئيسية :

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

int main(int argc, char** argv) {
    dbll myDoubleList = NULL;
    if (!init(myDoubleList)) {
        fprintf(stderr, "Insufficient Memory\n");
        return EXIT_FAILURE;
    } else {
        printf("length of the list = %d\nPrevious pointer = %p\nNext pointer = %p",
                myDoubleList->lengthOfTheList, myDoubleList->first, myDoubleList->last);
        return EXIT_SUCCESS;
    }
}
```

الدالتان printf و fprintf موجودتان في المكتبة stdio.h و الماكرو EXIT_FAILURE و EXIT_SUCCESS موجود في المكتبة stdlib.h و النوع bool مُعرف في المكتبة stdbool.h لذا قمنا باستدعاء المكتبات الثلاثة معاً.

في بداية الدالة الرئيسية قمنا بالإعلان عن قائمة جديدة و أسندنا لها العنوان NULL وهذه الخطوة ضرورية جدا في تهيئة المؤشرات بغض النظر عما تُشير إليه.

بعد ذلك, قمنا بتمرير القائمة myDoubleList إلى الدالة init كوسيط ثم تحققنا من القيمة المُعادة من طرف الدالة, إذا كانت false سيتم إظهار رسالة تنبيه على الشاشة و إلا فسيتم إظهار البيانات الابتدائية للعقدة الجديدة.

و هذه صورة لمُخرجات الكود :

```
length of the list = 0
Previous pointer = 00000000
Next pointer 00000000
```

لاحظ أن طول القائمة يُساوي 0, البعض يرى أن الصفر غير مناسب في هذه الحالة لأنه على الأقل تحتوي القائمة حالياً على عُقدة واحدة و بالتالي يجب أن يكون طول القائمة يُساوي واحد و ليس صفر, هذه وجهة النظر الأولى.

وجهة النظر الثانية (و هي التي أميل إليها) يقول أصحابها أنه حتى لو كانت القائمة تحتوي حالياً على عُقدة واحدة فإن هذه العُقدة خالية من المعلومات و لا تُشكل عُقدة حقيقة تُؤثر على طول القائمة و بالتالي يُمكن تجاهلها. في معظم الحالات تتم العودة لاحقاً إلى هذه العُقدة لتعديل بياناتها إلى بيانات حقيقية عن طريق إدراج عدد جديد كقيمة للمتغير الصحيح الموجود في العقدة, و بعدها تتم زيادة طول القائمة بواحد.

بالنسبة للعناوين فمن الطبيعي جداً أن تظهر الأصفار كقيم لعناوين المؤشرات لأن المؤشر NULL يملك العنوان 0x00000000 وفي العادة يكون هذا العنوان Invalid Memory Address في أغلب نظم التشغيل. يُستخدم الـ Null Pointer للدلالة على أن المؤشر فارغ أي لم يتم حجز ذاكرة له بعد.

إضافة عقدة جديدة

لإضافة عقدة جديدة إلى القائمة, يجب أن نمر بالخطوات التالية:

- حجز مساحة من الذاكرة للعقدة الجديدة.
- تهيئة كافة عناصر العقدة.
- المؤشر السابق للعقدة الجديدة سيُشير إلى NULL.
- المؤشر اللاحق للعقدة الجديدة سيُشير إلى بداية القائمة.
  - إذا كانت القائمة غير فارغة نجعل المؤشر السابق لبداية القائمة يُشير إلى العُقدة الجديدة.
  - في الحالة المُعاكسة نجعل المؤشر last يُشير إلى العُقدة الجديدة.
- نقوم بتحديث القائمة من خلال جعل العُقدة الجديدة هي الأولى.
- نزيد طول القائمة بواحد.

و بالتالي, دالة الإضافة ستكون هكذا:

```
bool addToTheTopOfTheList(dbll &argDbll, int value) {
    list newNode;
    if ((newNode = (list) malloc(sizeof (doublyLinkedList))) == NULL)
        return false;
    else {
        newNode->elem = value;
        newNode->ptrPrev = NULL;
        newNode->ptrNext = argDbll->first;
        argDbll->first
                ? argDbll->first->ptrPrev = newNode
                : argDbll->last = newNode;
        argDbll->first = newNode;
        argDbll->lengthOfTheList++;
        return true;
    }
}
```

كالعادة, إذا لم تنجح عملية الحجز ستُعيد الدالة false و إلا فالقيمة المُعادة ستكون true, هذا من جهة. من جهة أخرى, قمنا بتطبيق الخطوات التي ذكرناها آنفا, لا أكثر و لا أقل :-)

نأتي الآن إلى تضمين المكتبات اللازمة بالإضافة إلى تعليق موجز حول الـ main :

```
int main(int argc, char** argv) {
    dbll myDoubleList = NULL;
    init(myDoubleList);
    if (!addToTheTopOfTheList(myDoubleList, 7)) {
        fprintf(stderr, "Insufficient Memory\n");
        return EXIT_FAILURE;
    } else {
        display(myDoubleList);
        return EXIT_SUCCESS;
    }
}
```

في البداية, قمنا بتهيئة القائمة المزدوجة ثم استدعينا دالة الإضافة و تحققنا من القيمة المُعادة كما فعلنا سابقا مع دالة التهيئة, إذا تمت إعادة false نُظهر رسالة الخطأ و إلا فسنستدعي الدالة display المسئولة عن إظهار بيانات القائمة (سنشرح هذه الدالة لاحقاً).

مُخرجات الكود ستكون على هذا النحو:

```
elem(1) = 7

------oOo_ Fin _oOo------
```

لاحظ أنه لم يتم إظهار بيانات العقدة الفارغة لأنها مُحاطة بعناوين فارغة و بالتالي ستتوقف الدالة عن الإظهار عندما تصل إليها.

من ناحية أخرى, إدراج العُقدة الجديدة حدث قبل العُقدة الأولى و هذا ما يُسمى بالإضافة في بداية القائمة, يُمكننا أيضا أن نُضيف العُقدة في نهاية القائمة أو في أي مكان آخر و هنا تكمن أحد أبرز نقاط القوة لدى القوائم و هي المرونة.

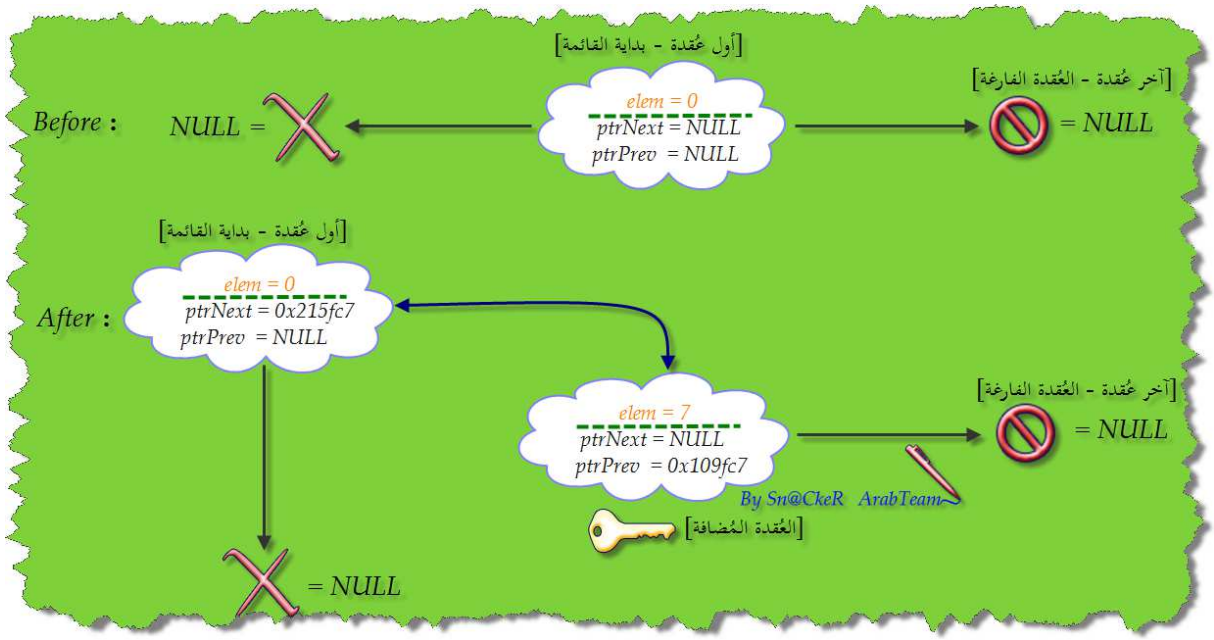
بالنسبة لإضافة العُقدة في نهاية القائمة فستكون هكذا:

```
bool addAtTheEndOfTheList(dbll &argDbll, int value) {
    list newNode;
    if ((newNode = (list) malloc(sizeof (doublyLinkedList))) == NULL)
        return false;
    else {
        newNode->elem = value;
        newNode->ptrPrev = argDbll->last;
        newNode->ptrNext = NULL;
        argDbll->last
                ? argDbll->last->ptrNext = newNode
                : argDbll->first = newNode;
        argDbll->last = newNode;
        argDbll->lengthOfTheList++;
        return true;
    }
}
```

نفس الخطوات السابقة مع تعديل طفيف يتمثل في :

- جعل المؤشر السابق للعقدة الجديدة يُشير إلى نهاية القائمة.
- إسناد القيمة NULL إلى المؤشر اللاحق للعقدة الجديدة.
  - إذا كانت القائمة غير فارغة نجعل المؤشر اللاحق لنهاية القائمة يُشير إلى العُقدة الجديدة.
  - في الحالة المُعاكسة نجعل المؤشر first يُشير إلى العُقدة الجديدة.
- نقوم بتحديث القائمة من خلال جعل العُقدة الجديدة هي آخر عُقدة.
- نزيد طول القائمة بواحد.

الصورة التالية تُوضح مفهوم الإضافة في نهاية القائمة المزدوجة:



يُمكننا أيضا إضافة عقدة جديدة في أي مكان من القائمة, لنفترض مثلا أننا نريد إدراج عقدة جديدة بعد عقدة أخرى يتم تحديد رقمها في القائمة. في هذه الحالة ستكون الخُطوات المُتبعة هي:

- الانتقال إلى العُقدة المحددة, توجد حالتان:
  - إذا وصلنا إلى NULL فهذا يعني أن رقم العقدة غير موجود, حينها نقوم بإعادة false.
  - في الحالة المعاكسة نقوم بالآتي:
- حجز مساحة من الذاكرة للعقدة الجديدة.
- تهيئة كافة عناصر العقدة.
- جعل المؤشر اللاحق للعقدة المحددة يُشير إلى المؤشر السابق للعقدة الجديدة.
- جعل المؤشر السابق للعقدة الجديدة يُشير إلى المؤشر اللاحق للعقدة المحددة.
- جعل المؤشر اللاحق للعقدة الجديدة يُشير إلى المؤشر الذي يُشير إليه المؤشر اللاحق للعقدة المحددة.
- جعل المؤشر السابق للعقدة التي تلي العقدة المحددة يُشير إلى المؤشر اللاحق للعقدة الجديدة.
- المؤشر first لن يتغير.
- المؤشر last يتغير فقط إذا كانت العُقدة المحددة هي آخر عقدة.
- زيادة طول القائمة بواحد.

و بالتالي دالة الإضافة ستكون هكذا:

```
bool addAfterACertainNode(dbll &argDbll, int value, int num) {
    list newNode, currentPointer = argDbll->first;
    if (num > argDbll->lengthOfTheList) return false;
    for (int i = 1; i < num; i++)
        currentPointer = currentPointer->ptrNext;
    if (currentPointer == NULL) return false;
    else {
        if ((newNode = (list) malloc(sizeof (doublyLinkedList))) == NULL)
            return false;
        newNode->elem = value;
        newNode->ptrPrev = currentPointer;
        newNode->ptrNext = currentPointer->ptrNext;
        (currentPointer->ptrNext == NULL)
                ? argDbll->last = newNode
                : currentPointer->ptrNext->ptrPrev = newNode;
        currentPointer->ptrNext = newNode;
        argDbll->lengthOfTheList++;
        return true;
    }
}
```

إذا لم تنجح عملية الحجز أو لم يتم العثور على رقم العقدة ستُعيد الدالة false و إلا فستعيد الدالة true بعد أن تتم إضافة العُقدة الجديدة و تحديث القائمة.

الآن أصبح من السهل جداً كتابة دالة تقوم بتعديل بيانات عقدة مُعينة, فقط بدلا من إدراج عُقدة جديدة, نقوم بتغيير بيانات العُقدة التي توقفت عندها الحلقة for ما لم تكن تلك العقدة فارغة.

هذا مثال على استدعاء الدالة addAfterACertainNode في الـ main :

```
int main(int argc, char** argv) {
    dbll myDoubleList = NULL;
    init(myDoubleList);
    addToTheTopOfTheList(myDoubleList, 2);
    addToTheTopOfTheList(myDoubleList, 1);
    addAtTheEndOfTheList(myDoubleList, 4);
    addAtTheEndOfTheList(myDoubleList, 5);
    if (!addAfterACertainNode(myDoubleList, 3, 2)) {
        fprintf(stderr, "Insufficient Memory Or Node Not Found\n");
        return EXIT_FAILURE;
    } else {
        display(myDoubleList);
        return EXIT_SUCCESS;
    }
}
```

يُمكننا جعل الدالة addAfterACertainNode تعيد int و نخصص قيمة لكل خطأ فمثلا يمكننا اعتبار أن الصفر يدل على عدم وجود العقدة و 1 يدل على فشل عملية الحجز و 2 تدل على نجاح عملية الإدراج كما يُمكننا أيضا جعل الدالة تعيد void و في هذه الحالة يُستحسن إضافة رسائل الخطأ داخل جسم الدالة.

على كل حال, المُخرجات ستكون هكذا:

```
elem(1) = 1
elem(2) = 2
elem(3) = 3
elem(4) = 4
elem(5) = 5


------oOo_ Fin _oOo------
```

نأتي الآن إلى شرح كيفية عمل دالة الإظهار التي تقوم بعرض محتويات القائمة :

```
void display(dbll &argDbll) {
    list currentPointer;
    int i = 1;
    currentPointer = argDbll->first;
    while (currentPointer) {
        printf("elem(%d) = %d\n", i++, currentPointer->elem);
        currentPointer = currentPointer->ptrNext;
    }
    printf("\n\n------oOo_ Fin _oOo------\n");
}
```

حتى لا نفقد بيانات القائمة, قمنا بتخزين نسخة من عنوان العقدة الأولى داخل المؤشر currentPointer و مادام هذا الأخير غير فارغ, سيتم إظهار العدد الموجود في العقدة الحالية و الانتقال إلى العقدة الموالية. بالنسبة للمتغير i فتأتي فائدته في ترقيم عناصر العقدة.

حذف عقدة مُعينة

كما فعلنا سابقا مع عملية الإضافة, يمكننا أيضا حذف عُقدة من البداية, النهاية أو أي مكان آخر من القائمة.

فمثلا, الدالة التالية تقوم بحذف آخر عُقدة من القائمة :

```
bool removeTheLast(dbll &argDbll) {
    list tmpNode = argDbll->last;
    if (!tmpNode) return false;
    argDbll->last->ptrPrev->ptrNext = NULL;
    argDbll->last = argDbll->last->ptrPrev;
    argDbll->lengthOfTheList--;
    free(tmpNode);
    return true;
}
```

الخُطوات المُتبعة هي:

- نُخزن عنوان آخر عقدة في متغير مؤقت.
- إذا كانت القائمة فارغة, تتم إعادة false و ينتهي الأمر.
- في الحالة المعاكسة نقوم بالخطوات التالية:
  - نجعل المؤشر اللاحق للعقدة القبل الأخيرة يُشير إلىNULL.
  - نقوم بتحديث القائمة من خلال جعل العُقدة القبل الأخير هي آخر عقدة.
  - ننقص طول القائمة بواحد.
  - نُحرر العنوان الذي كان يُشير إلى العُقدة الأخيرة.

هذا مثال على استدعاء الدالة removeTheLast في الـ main :

```c
int main(int argc, char** argv) {
    dbll myDoubleList = NULL;
    init(myDoubleList);
    addToTheTopOfTheList(myDoubleList, 2);
    addToTheTopOfTheList(myDoubleList, 1);
    addAtTheEndOfTheList(myDoubleList, 4);
    addAfterACertainNode(myDoubleList, 3, 2);
    addAtTheEndOfTheList(myDoubleList, 5);
    printf("Before :\n");
    display(myDoubleList);
    if (!removeTheLast(myDoubleList)) {
        fprintf(stderr, "Error - Stack Empty\n");
        return EXIT_FAILURE;
    } else {
        printf("\nAfter :\n");
        display(myDoubleList);
        return EXIT_SUCCESS;
    }
}
```

إذا كانت القائمة فارغة سيتم إظهار رسالة الخطأ, في الحالة المعاكسة سيتم إظهار عناصر القائمة قبل و بعد الحذف.

المُخرجات ستكون هكذا:

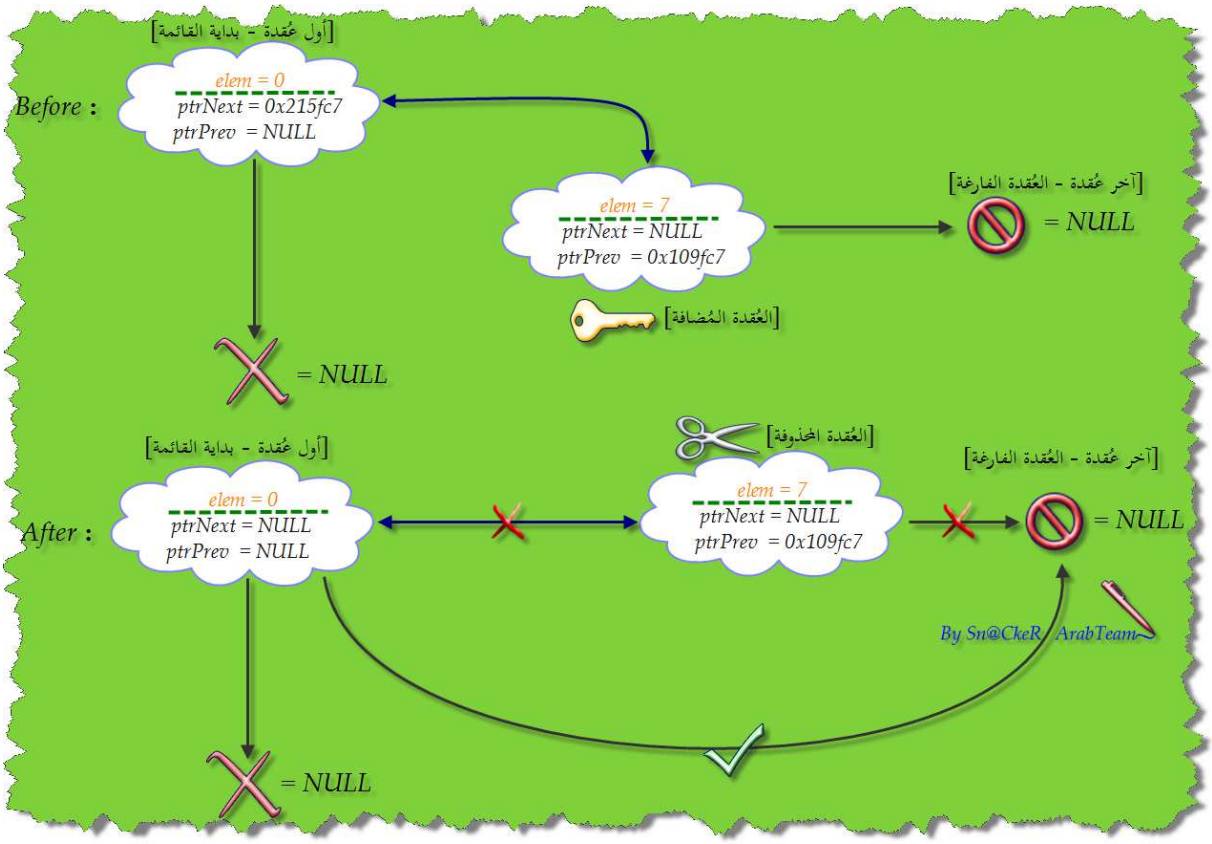
```
Before :
elem(1) = 1
elem(2) = 2
elem(3) = 3
elem(4) = 4
elem(5) = 5


------oOo_ Fin _oOo------

After :
elem(1) = 1
elem(2) = 2
elem(3) = 3
elem(4) = 4


------oOo_ Fin _oOo------
```

الصورة التالية تُوضح مفهوم حذف آخر عُقدة في قائمة مزدوجة:



أما إذا أردنا حذف عقدة من بداية القائمة فستكون الخطوات كما يلي:

- تخزين عنوان العقدة الأولى داخل متغير مؤقت.
- جعل مؤشر القائمة يُشير إلى العقدة الثانية.
- تحرير المؤشر الذي كان يُشير إلى العقدة الأولى.

و بالتالي دالة الحذف في هذه الحالة ستكون هكذا:

```
bool removeTheFirst(dbll &argDbll) {
    list tmpNode = argDbll->first;
    if (!tmpNode) return false;
    argDbll->first = argDbll->first->ptrNext;
    argDbll->first->ptrPrev = NULL;
    argDbll->lengthOfTheList--;
    free(tmpNode);
    return true;
}
```

لاحظ أنه تم التأكد أن القائمة غير فارغة قبل إجراء عملية الحذف و هذا مُهم جدا إذْ يجب التأكد من مثل هذه الحالات قبل القدوم على تنفيذ العملية.

نأتي الآن إلى كيفية الحذف عن طريق تحديد قيمة المتغير elem في العقدة المراد حذفها :

```
bool removeAfterACertainNode(dbll &argDbll, int num) {
    list tmpNode, currentPointer = argDbll->first;
    while (currentPointer) {
        if (currentPointer->elem == num) break;
        else currentPointer = currentPointer->ptrNext;
    }
    if (currentPointer == NULL) return false;
    else {
        tmpNode = currentPointer;
        currentPointer->ptrPrev->ptrNext = currentPointer->ptrNext;
        currentPointer->ptrNext->ptrPrev = currentPointer->ptrPrev;
        free(tmpNode);
        argDbll->lengthOfTheList--;
        return true;
    }
}
```

هذه المرة قمنا بالحذف على أساس العقدة التي تحمل قيمة معينة لـ elem و ليس على أساس رقم العقدة في القائمة.

الدالة السابقة بإمكانها حذف أي عُقدة من القائمة باستثناء العقدة الأولى و الأخيرة, يُمكننا إضافة شروط للتحقق من مكان العقدة فإذا كانت العقدة المراد حذفها هي العقدة الأولى نقوم باستدعاء الدالة removeTheFisrst و removeTheLast إذا كانت آخر عقدة.

بالنسبة لتفاصيل الدالة فعند الخروج من while سنكون أمام خيارين, الخيار الأول يعني أنه لا توجد عقدة تحمل القيمة elem التي تم إرسالها للدالة, حينها سنقوم بإعادة false و تنتهي المهمة, إذا تجاوزنا الـ if بسلام فهذا يعني أنه توجد عُقدة تحمل قيمة elem داخلها, لذا سنقوم بتخزين العقدة التي تُحقق الشرط في المؤشر tmpNode ثم نُحرر عنوان تلك العُقدة بعد أن نربط بين العقدتين اللتين يحيطان بالعقدة المراد حذفها.

حساب طول القائمة

هذه المرة, لن نحتاج إلى كتابة دالة منفردة لحساب الطول, لأن طول القائمة مُخزن في المتغير lengthOfTheList و يتم تحديث قيمته كلما تمت إضافة أو حذف عقدة معينة.

دمج قائمتين في قائمة واحدة

دالة الدمج تستقبل وسيطين يُمثلان القائمتين المُراد دمجهما :

```
dbll mergeTwoLists(dbll firstList, dbll secondList) {
    if (!firstList) return secondList;
    else {
        firstList->last->ptrNext = secondList->first;
        return firstList;
    }
}
```

إذا كانت القائمة الأولى فارغة, فهذا يعني أن دمج القائمتين يُعطي القائمة الثانية لذا تمت إعادتها. في الحالة المُعاكسة نقوم بربط آخر عُقدة من القائمة الأولى بأول عُقدة من القائمة الثانية.

حذف القائمة

دالة الحذف فكرتها كالآتي : في كل مرة نقوم بتخزين عنوان العقدة الحالية في مؤشر مؤقت ثم ننتقل إلى العقدة الموالية و نحرر المؤشر و هكذا حتى نصل إلى عنوان فارغ (NULL) مما يعني أن القائمة انتهت و بالتالي نخرج من الحلقة while.

ثم نقوم بتحديث عناصر المصفوفة المساعدة (نجعل المؤشرات تُشير إلى NULL و نُسند القيمة صفر إلى المتغير الذي يحوي طول القائمة).

```
void clearTheList(dbll &argDbll) {
    list tmpNode, currentPointer = argDbll->first;
    while (currentPointer) {
        tmpNode = currentPointer;
        currentPointer = currentPointer->ptrNext;
        free(tmpNode);
    }
    argDbll->first = NULL;
    argDbll->last = NULL;
    argDbll->lengthOfTheList = 0;
}
```

اختبر قدراتك

لدينا زجاجة تحتوي على 7 كرات, واحدة باللون الأحمر و اثنتين باللون الأصفر و أربعة باللون الأخضر. يقوم اللاعب بسحب كرة عشوائيا من الزجاجة, إذا كانت الكرة حمراء سيربح اللاعب 10 دولار و يخسر 5 دولار إذا كانت صفراء أما إذا كانت خضراء فسيحظى اللاعب بسحب كرة أخرى من الزجاجة (دون أن يعيد الكرة الأولى التي سحبها), إذا كانت الكرة الجديدة حمرءا سيربح 8 دولار و إلا سيخسر 4 دولار. تنتهي اللعبة عندما يصبح رصيد اللاعب يساوي صفر علما أن الرصيد الابتدائي لكل لاعب يُساوي 6 دولار.

قم بعمل برنامج صغير يُحاكي هذه اللعبة باستخدام القوائم المزدوجة.

الجزء الثالث – المكدسات (Stacks)

تعريف

يُمكن تشبيه المكدس أو الـ Stack بمجموعة الصحون حيث يُمكننا إضافة صحن في القمة لكن عندما نريد سحب أحد الصحون, يلزمنا سحب كافة الصحون الموجودة فوقه و هذا النوع من الترتيب يُعرف اختصاراً بـ LIFO أي Last In First Out.

المكدس لا يملك نوع بيانات خاص به و إنما هو مجرد تركيبة يُمكن محاكاتها باستخدام المصفوفات أو القوائم (سواء كانت بسيطة أو مزدوجة) أو حتى الأشجار (Trees).

المحاكاة باستخدام المصفوفات

لمحاكاة المكدس بالمصفوفات سنحتاج إلى :

- عدد ثابت يُمثل العدد الأقصى لعناصر المكدس.
- مصفوفة لتخزين العناصر.
- و متغير صحيح يُمثل الـ index الخاص بقمة المكدس.

إذا الإعلان عن العناصر السابقة سيكون هكذا:

```
#define MAXSIZE 20
int stack[MAXSIZE];
int top = 0;
```

نأتي الآن إلى دالة الإدراج :

```
void push() {
    int num;
    if (top >= MAXSIZE) {
        printf("STACK FULL");
        return;
    } else {
        if (top < 0)
            top = 0;
        printf("ENTER THE STACK ELEMENT : ");
        scanf("%d", &num);
        stack[top++] = num;
    }
}
```

إذا كانت قيمة top أكبر أو تساوي من MAXSIZE فهذا يعني أن المكدس قد امتلأ لذا قمنا بإظهار رسالة تفيد بذلك.

في الحالة المعاكسة, إذا كانت قيمة top أقل من الصفر, نقوم بإعادتها للصفر ثم نقرأ القيمة التي أدخلها المستخدم و نخزنها في قمة المكدس ثم نجعل المتغير top يُشير إلى الخانة الموالية.

بالنسبة لدالة الحذف فهي كالتالي :

```
void pop() {
    if (top >= 0)
        top--;
}
```

إذا كان الـ index الذي يُشير إلى قمة المكدس أكبر أو يساوي صفر نقوم بعمل decrement له مما يعني الرجوع إلى الخلف بخطوة واحدة.

دالة الإظهار هي التي تُظهر حقيقة المحاكاة :-)

```c
void display() {
    if (top <= 0)
        printf("STACK EMPTY");
    else {
        printf("-->TOP ");
        for (int i = top - 1; i >= 0; i--)
            printf("%d\n", stack[i]);
    }
}
```

القيمة الابتدائية للمتغير top هي 0 و بالتالي إذا كان top أقل أو يُساوي صفر فهذا يعني أن المكدس فارغ.

في الحالة المعاكسة, سنقوم بإظهار العناصر ابتداء من قمة المكدس وصولا إلى الصفر. (في الحقيقة, ما يحدث هو أن المكدس عبارة عن مصفوفة و إظهار العناصر يتم بالمقلوب أي من الأخير إلى الأول)

المحاكاة باستخدام القوائم – مُقدمة

يمكننا اعتبار أن المكدس ما هو إلا حالة خاصة من القوائم المتصلة حيث لا يُمكن إضافة أو حذف عنصر إلا من بداية القائمة لذا فإن العمليات ستقتصر أساسا على دالتين أساسيتين هما:

- ✓ دالة اسمها Push تقوم بإدراج عنصر في بداية المكدس.
- ✓ دالة اسمها Pop تقوم بحذف عنصر من بداية المكدس.

المحاكاة باستخدام القوائم البسيطة

الإعلان عن المكدس سيكون هكذا:

```c
struct myStack {
    int value;
    struct myStack *ptrPrev;
};
struct myStack *top, *temp;
```

المكدس سيحتوي على قيمة واحدة من نوع int بالإضافة إلى المؤشر ptrPrev الذي يُمثل مفتاح الدخول. المؤشر top يُشير إلى قمة المكدس و المؤشر temp عبارة عن مؤشر مؤقت.

نبدأ مع دالة التهيئة :

```
void create() {
    top = (struct myStack *) malloc(sizeof (struct myStack));
    printf("ENTER THE FIRST ELEMENT: ");
    scanf("%d", &top->value);
    top->ptrPrev = NULL;
    temp = top;
}
```

في البداية, قمنا بحجز مساحة جديدة للمؤشر top ثم طلبنا من المستخدم إدخال عدد صحيح ليتم تخزينه في المتغير value داخل العقدة الجديدة. المؤشر ptrPrev جعلناه يُشير إلى NULL و temp إلى top.

دالة الإضافة مُشابهة جدا لدالة التهيئة :

```
void push() {
    top = (struct myStack *) malloc(sizeof (struct myStack));
    printf("ENTER THE FIRST ELEMENT: ");
    scanf("%d", &top->value);
    top->ptrPrev = temp;
    temp = top;
}
```

فقط, الفرق يكمن في كيفية ربط المؤشر ptrPrev بالمؤشر الموالي.

دالة الحذف ستكون كالآتي :

```
void pop() {
    if (temp == NULL) {
        printf("STACK IS EMPTY");
    } else {
        top = temp;
        printf("DELETED ELEMENT IS %d", temp->value);
        temp = temp->ptrPrev;
        free(top);
    }
}
```

إذا كان temp يُشير إلى NULL فهذا يعني أن المكدس فارغ لذا قمنا بإظهار رسالة تُفيد بذلك.

في الحالة المعاكسة سنقوم بنسخ temp داخل top و نُظهر القيمة التي سيتم حذفها ثم نحرر المؤشر top بعد أن ننتقل إلى العقدة السابقة.

بالنسبة لدالة الإظهار فلن تتغير (سبق و أن شرحناها في الجزء الأول) :

```
void display() {
    top = temp;
    while (top != NULL) {
        printf("%d\n", top->value);
        top = top->ptrPrev;
    }
}
```

المحاكاة باستخدام القوائم المزدوجة

الإعلان عن مكدس باستخدام القوائم المزدوجة سيكون قريبا جدا من الإعلان السابق, فقط نُضيف مؤشر لاحق :

```
struct myStack {
    int value;
    struct myStack *ptrNext;
    struct myStack *ptrPrev;
};

typedef struct myStack *stack;
```

دالة الإضافة ستكون كالآتي :

```c
void push(stack *myStack) {
    stack newStack = (stack) malloc(sizeof (myStack));
    printf("Enter element to push\n");
    scanf("%d", &newStack->value);
    if (*myStack == NULL) {
        newStack->ptrNext = newStack->ptrPrev = NULL;
        *myStack = newStack;
    } else {
        newStack->ptrNext = *myStack;
        (*myStack)->ptrPrev = newStack;
        newStack->ptrPrev = NULL;
        *myStack = newStack;
    }
}
```

في البداية, قمنا بالإعلان عن مكدس جديد باسم newStack و قمنا بحجز المساحة المطلوبة له ثم طلبنا من المستخدم إدخال قيمة و قمنا بتخزينها في المتغير value.

إذا كان newStack يُشير إلى NULL فهذا يعني أن المكدس فارغ و بالتالي العقدة الجديدة ستُحاط بعناوين NULL من كلا الجانبين. في الحالة المعاكسة سنجعل المؤشر اللاحق لـ newStack يُشير إلى myStack و المؤشر السابق لـ myStack يُشير إلى newStack و المؤشر السابق لـ newStack يُشير إلى NULL ثم نقوم بتحديث المكدس.

بالنسبة لدالة الحذف فستكون كالتالي :

```c
void pop(stack *myStack) {
    stack del;
    if (*myStack == NULL) {
        printf("Stack is Empty ..\n");
        return;
    }
    printf("Deleted .. %d\n", (*myStack)->value);
    del = *myStack;
    *myStack = (*myStack)->ptrNext;
    free(del);
    if ((*myStack)) {
        (*myStack)->ptrPrev = NULL;
    }
}
```

إذا كان المكدس فارغ سيتم إظهار رسالة تُفيد بذلك و الخروج من الدالة.

في الحالة المعاكسة, سيتم إظهار العنصر الذي سيتم حذفه و التقدم إلى العقدة الموالية و حذف المؤشر الذي يُشير إلى العقدة السابقة.

عند الانتهاء من عملية الحذف, نتحقق من ما إذا كان المكدس يحتوي على عقدة واحدة أم لا ؟ إذا كان الجواب نعم, نقوم بربط العقدة الوحيدة بالمؤشر NULL كمؤشر سابق لها.

دالة الإظهار لن تتغير (سبق و أن شرحناها في الجزء الثاني) :

```
void display(stack myStack) {
    stack temp = myStack;
    printf("Elements are ...\n");
    while (temp) {
        printf("%d\n", temp->value);
        temp = temp->ptrNext;
    }
}
```

اختبر قدراتك

قم بعمل برنامج يحول infix إلى postfix باستخدام الـ Stacks.

الجزء الرابع – الطوابير (Queues)

تعريف

الطابور أو الـ Queue مُشابه جدا للمكدس (Stack) فقط الفرق الوحيد بينهما يكمن في أن الإضافة في الطابور تكون في النهاية بينما تكون في البداية عند الحديث عن المكدس, في بقية العمليات (الحذف و الإظهار) لا يوجد فرق يُذكر.

أقرب مثال على الطابور هو قوائم الانتظار أو الـ Waiting List حيث نجد أن أول من يدخل هو أول من يخرج و آخر من يدخل هو آخر من يخرج و هذا النوع من الترتيب يُعرف اختصاراً بـ FIFO أي First In First Out.

كما هو الحال مع المكدس, الطابور لا يملك نوع بيانات خاص به و إنما هو مجرد تركيبة يُمكن محاكاتها باستخدام المصفوفات أو القوائم (سواء كانت بسيطة أو مزدوجة) أو حتى الأشجار (Trees).

المحاكاة باستخدام المصفوفات

لمحاكاة الطابور بالمصفوفات سنحتاج إلى 3 متغيرات: الأول عبارة عن المصفوفة التي ستُحاكي الطابور و المتغير الثاني يحمل رقم أول خانة من الطابور و المتغير الثالث يحمل رقم آخر خانة, هكذا:

```
#define SIZE 20
int Queue[SIZE], front, rear = front = -1;
```

بالنسبة لتهيئة المتغيرات, قمنا بحجز 20 خانة للمصفوفة Queue ثم أسندنا القيمة 1- لكل من front و rear كدلالة على فراغ الطابور.

نأتي الآن إلى دالة الإدراج :

```
void push() {
    if (rear == (SIZE - 1)) {
        printf("Overflow!");
    } else {
        rear++;
        printf("Enter element: ");
        scanf("%d", &Queue[rear]);
    }
}
```

إذا تساوت قيمة rear مع SIZE-1 فهذا يعني أننا وصلنا إلى آخر خانة و بالتالي لا يمكننا إضافة المزيد من العناصر لذا قمنا بإظهار الرسالة OverFlow كدلالة على حدوث فيض عند محاولة الإدراج.

في الحالة المعاكسة, سنقوم بالانتقال إلى الخانة الموالية ثم نقرأ العدد المدخل و نخزنه في الخانة الحالية.

بالنسبة لدالة الحذف فهي كالتالي :

```
void pop() {
    if (front == rear) {
        printf("Underflow!");
    } else {
        printf("Element popped: %d\n", Queue[++front]);
    }
}
```

إذا تساوت قيمة rear مع front فهذا يعني أنه لا يوجد عنصر للسحب, هذا من جهة.

من جهة أخرى, يتساوى rear مع front عند -1 فقط و من المعروف أن -1 لا يمكن أن تكون index لأحد عناصر المصفوفة لأن الترقيم يبدأ من 0 لذا قمنا بإظهار الرسالة UnderFlow كدلالة على أن index المصفوفة أقل من الصفر.

في الحالة المعاكسة, سنقوم بإظهار قيمة الخانة الحالية و الانتقال إلى الخانة الموالية (لذا أجد أن هذه المحاكاة سيئة جداً لأنه لا يتم حذف العناصر بصفة حقيقية كتحرير الذاكرة كما يحدث في القوائم).

دالة الإظهار أعتقد أنها واضحة, إذا تساوت قيمة rear مع front فهذا يعني أن الطابور فارغ. في الحالة المعاكسة, سنقوم بإظهار كافة عناصر الطابور :

```
void display() {
    if (front == rear) {
        printf("Queue Empty");
    } else {
        for (int i = (front + 1); i <= rear; i++) {
            printf("%d ", Queue[i]);
        }
    }
}
```

المحاكاة باستخدام القوائم – مُقدمة

كما قلنا سابقا, توجد عدة صفات مشتركة بين الطابور و المكدس و بالتالي يمكننا اعتبار أن الطابور ما هو إلا حالة خاصة من القوائم المتصلة حيث لا يُمكن إضافة عنصر إلا في نهاية القائمة أما عملية الحذف فتظل كما كانت.

بالرغم من تشابه الـ Queue و الـ Stack إلا أن التغييرات ستكون مُعقدة نسبيا نظرا لأن الطابور يُلزمنا بالانتقال إلى بداية أو نهاية القائمة حسب نوع العملية المطلوب إجرائها (إضافة أو حذف).

المحاكاة باستخدام القوائم البسيطة

الإعلان عن الطابور سيكون هكذا:

```
struct myQueue {
    int value;
    struct myQueue *ptrNext;
} *front, *rear;
```

الطابور يحتوي على قيمة واحدة من نوع int بالإضافة إلى المؤشر ptrNext الذي يُمثل مفتاح الدخول للعقدة الموالية. المؤشر front يُشير إلى قمة الطابور بينما يُشير rear إلى مؤخرة الطابور.

نبدأ مع دالة الإضافة :

```c
void enqueue() {
    struct myQueue *q;
    q = (struct myQueue*) malloc(sizeof (struct myQueue));
    printf("Enter The Element Value : ");
    scanf("%d", &q->value);
    q->ptrNext = NULL;
    if (rear == NULL || front == NULL)
        front = q;
    else
        rear->ptrNext = q;
    rear = q;
}
```

في البداية, قمنا بحجز مساحة جديدة للمؤشر q ثم طلبنا من المستخدم إدخال عدد صحيح ليتم تخزينه في المتغير value داخل العقدة الجديدة. إذا كان أحد المؤشرين front أو rear يُشير إلى NULL فهذا يعني أن الطابور فارغ لذلك ستلعب دالة الإضافة في هذه الحالة دور دالة التهيئة, في الحالة المعاكسة سنجعل المؤشر اللاحق لـ rear يُشير إلى الطابور ثم نقوم بتحديث المؤشر rear.

أما دالة الحذف فهي كالآتي :

```c
int dequeue() {
    struct myQueue *q;
    if (front == NULL || rear == NULL)
        printf("Under Flow");
    else {
        q = front;
        printf("The deleted element = %d\n", q->value);
        front = front->ptrNext;
        free(q);
    }
}
```

إذا كان حمل أحد المؤشرين front أو rear القيمة NULL فهذا يعني أن الطابور فارغ و بالتالي لا توجد عناصر للسحب لذا قمنا بإظهار رسالة تُفيد بذلك.

في الحالة المعاكسة سنقوم بنسخ front داخل q و نُظهر القيمة التي سيتم حذفها ثم نحرر المؤشر q بعد أن ننتقل إلى العقدة الموالية.

بالنسبة لدالة الإظهار فلن تتغير كثيرا (سبق و أن شرحنا فكرتها في الجزء الأول) :

```c
void display() {
    struct myQueue *t;
    t = front;
    while (front == NULL || rear == NULL) {
        printf("Queue is empty");
    }
    while (t != NULL) {
        printf("->%d", t->value);
        t = t->ptrNext;
    }
}
```

المحاكاة باستخدام القوائم المزدوجة

لتمثيل الطابور باستخدام القوائم المزدوجة يكفي أن نتذكر معا كيفية الإعلان عن قائمة مزدوجة بسيطة تحوي عنصر واحد مثلا:

```c
#define MAXSIZE  20
struct myQueue {
    int value;
    struct myQueue *ptrNext;
    struct myQueue *ptrPrev;
} *head, *tail;

int length = 0;
```

المؤشر head يُشير إلى قمة الطابور بينما يُشير tail إلى مؤخرة الطابور أما المتغير length فيُمثل طول الطابور.

دالة الإضافة ستكون كالآتي :

```
void enqueue(int x) {
    if (length > MAXSIZE) {
        printf("Queue Overflow");
        return;
    }
    myQueue *temp
            = (struct myQueue*)malloc(sizeof(struct myQueue));
    temp->value = x;
    temp->ptrNext = NULL;

    if (length == 0) {
        temp->ptrPrev = NULL;
        temp->ptrNext = NULL;
        head = temp;
    } else {
        temp->ptrPrev = tail;
        tail->ptrNext = temp;
    }
    tail = temp;
    length++;
}
```

في البداية, قمنا بالتحقق من أن الطابور لم يمتلئ بعد, إذا كان قد امتلأ نُظهر رسالة تفيد بذلك و يتم الخروج من الدالة.

إذا تجاوزنا مرحلة التحقق فهذا يعني أن الطابور لم يمتلأ بعد لذا قمنا بالإعلان عن مكدس جديد باسم temp و قمنا بحجز المساحة المطلوبة له ثم أسندنا قيمة الوسيط إلى المتغير value و جعلنا المؤشر اللاحق يُشير إلى NULL.

إذا كان طول الطابور يُساوي صفر فهذا يعني أن العقدة المراد إضافتها هي أول عقدة لذا جعلنا المؤشر السابق يُشير إلى NULL و كذلك اللاحق ثم جعلنا المؤشر head يُشير إلى بداية الطابور.

في الحالة المعاكسة (هذا يعني أنه توجد أكثر من عقدة) نجعل المؤشر السابق للعقدة الحالة يُشير إلى tail ثم نجعل المؤشر اللاحق لـ tail يُشير إلى العقدة الحالية.

في النهاية نقوم بتحديث الطابور ثم نزيد طول الطابور بواحد.

بالنسبة لدالة الحذف فستكون كالتالي :

```c
void dequeue() {
    if (length <= 0) {
        printf("Nothing can be deleted");
        return;
    }
    myQueue *temp = head;
    head = head->ptrNext;
    free(temp);
    temp = NULL;
    length--;
}
```

إذا كانت قيمة المتغير length أقل أو تُساوي صفر فهذا يعني أن الطابور فارغ و بالتالي سيتم إظهار رسالة تُفيد بذلك و الخروج من الدالة.

في الحالة المعاكسة, سنقوم بحفظ نسخة من المؤشر المراد تحريره ثم ننتقل إلى العقدة الموالية و نحرر المؤشر ثم نجعله يُشير إلى NULL (هذه الخطوة مهمة جدا و هي عادة حسنة عند التعامل مع المؤشرات) و أخيرا ننقص طول الطابور بواحد.

دالة الإظهار لن تتغير كثيراً (سبق و أن شرحناها في الجزء الثاني) :

```c
void display() {
    if (length <= 0) {
        printf("Queue Empty");
        return;
    } else {
        myQueue *temp = head;
        while (temp != NULL) {
            printf("%d ", temp->value);
            temp = temp->ptrNext;
        }
    }
}
```

اختبر قدراتك

اكتب برنامج يطلب من المستخدم إدخال جملة ثم يُخبره ما إذا كانت تلك الجملة تُمثل Palindrome أو لا.

ملاحظة : قم بتخزين الجملة في Stack ثم خزن نسخة أخرى من الجملة في Queue و قارن بين محتوى البنيتين.


تم بحمد الله