



كلية الهندسة قسم المعلوماتية

بنى معطيات 1

Data Structure 1

ا.د. علي عمران سليمان

محاضرات الأسبوع الخامس

الأرتال

QUEUES

الفصل الثاني 2023-2024

stacks	المكدرات
	1- مقدمة
	2- مدخل إلى الأرتال
	3- تحقيق الأرتال باستخدام المصفوفات
	4- تطبيقات على الأرتال
	اللوائح 1
	Lists
	1- مقدمة
	1- تحقيق اللوائح باستخدام التخزين المتتالي

References:

- ADTs, Data Structures, and Problem Solving with C++: International Edition, by Larry R. Nyhoff, Publisher: Pearson , 2004
- Data Structures and Algorithms in Java, 6 th, [Roberto Tamassia](#), [Michael T. Goodrich](#), [Michael H. Goldwasser](#), Pub. Wiley 2014
- د.علي سليمان، بني معطيات بلغة JAVA، بني معطيات بلغة C++، بني معطيات بلغة Pascal جامعة تشرين 2014، 2007، 1998

1- مقدمة:

الأرتال queues, وهي بنية شبيهة إلى حد بعيد ببنية المكدرات, وتستخدم في طيف واسع من التطبيقات (كما المكدرات). يمكن تحقيق الأرتال باستخدام المصفوفات كبنية تخزين أساسية, لكن وكما سنرى يحتاج تحقيقها إلى جهد أكبر قليلاً من المكدرات, كما سنرى في بعض تطبيقاتها.

نركز في هذا الفصل على بناء الصنف queues وسنرى في فصل لاحق بعض التحسينات على هذه البنية

وأهمها:

- تحويله إلى قالب صنف وبالتالي يصبح نمط حاوي عام يمكن أن يعالج أي نمط من عناصر الرتل ADT.
- استخدام vector لتخزين الحاوي وبالتالي سعة الرتل يمكن أن تكبر بحسب الحاجة.
- التعرف على الحاوي queue المعرف في مكتبة القوالب القياسية (STL (Standard Template Library).

لننظر إلى المسائل التالية:

يمكن تعريف الرتل queue على أنه صف انتظار.

- مثل صف من الناس ينتظرون الدفع على صندوق المشتريات .
- صف من العربات على حاجز عبور .
- رتل من الطائرات ينتظرون الهبوط في المطار .
- رتل من المهام في نظام حاسوبي ينتظرون استخدام جهاز خرج كالطابعة مثلاً.
- في كل من الأمثلة السابقة تتم خدمة العناصر في نفس الترتيب التي ترد فيه, أي أول عنصر في الرتل هو أول عنصر ستم خدمته .
- يعتمد الرتل على بنية تقوم على مبدأ الداخل أولاً يخرج أولاً "First-In-First-Out FIFO" أو القادم أولاً يخدم أولاً "First-Come-First-Served FCFS".
- كنمط بيانات مجرد ADT, يمكن النظر إلى الرتل على أنه نوع خاص من اللوائح, هذا النوع الخاص تكون فيه عمليات الإضافة والحذف مقيدة بطرفي اللائحة, على عكس المكس الذي ترتبط فيه عمليات الإضافة والحذف مقيدة بطرف واحد من اللائحة.
- يتم حذف العناصر من الرتل من طرف واحد يدعى بداية الرتل front (أو الرأس head) .
- تضاف العناصر فقط من الطرف الآخر الذي يدعى نهاية back (أو العجز rear أو الذيل tail). باقي العمليات الأساسية الأخرى تتمثل في بناء رتل فارغ , اختبار فيما إذا كان الرتل فارغاً .
- يمكن تلخيص تعريف الرتل كنمط بيانات مجرد كما يلي:

الرتل كنمط بيانات مجرد ADT

مجموعة مرتبة من عناصر البيانات تتمتع بخاصية أن العناصر يمكن أن تحذف فقط من طرف واحد يدعى بداية الرتل front, ويمكن أن تضاف العناصر فقط من الطرف الآخر ويدعى نهاية الرتل back.

العمليات الأساسية:

- بناء الرتل (عادة فارغ).
- اختبار فيما إذا كان الرتل فارغ (EmptyQ) .
- إضافة عنصر إلى نهاية الرتل (AddQ).
- معرفة العنصر الموجود في بداية الرتل (FrontQ).
- حذف العنصر الموجود في بداية الرتل (RemoveQ)

بما أن الأرتال تشبه إلى حد ما المكذسات, فإننا سنقوم بمحاكاة تحقيق المكذسات باستخدام المصفوفات المبين في الفصل السابق لإجراء تحقيق الأرتال باستخدام المصفوفات, وبالتالي سنستخدم:

myArray: لتخزين عناصر الرتل. #

متحولين من النوع الصحيح: #

i. myFront: الموقع في مصفوفة العناصر الذي يمكن أن تزال العناصر منه, أي موقع العنصر في مقدمة الرتل.

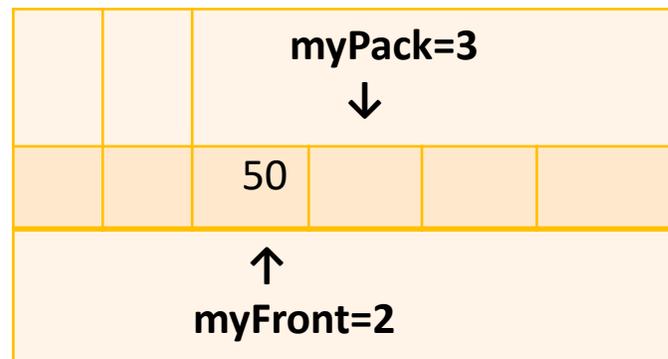
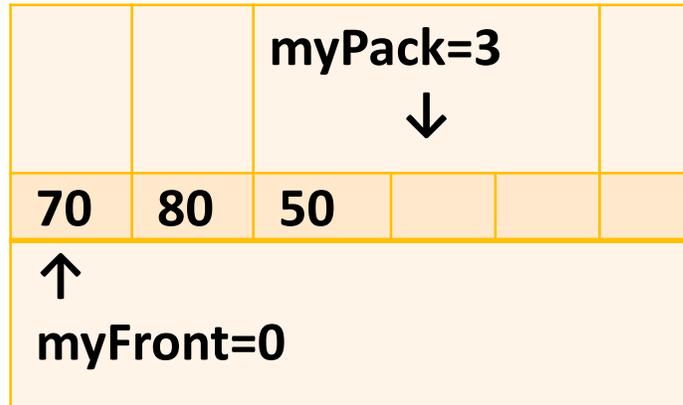
ii. myBack: الموقع في مصفوفة العناصر الذي يمكن أن تضاف العناصر إليه, أي موقع العنصر التالي لآخر عنصر في

الرتل.

وبالتالي يمكن أن نمثل هذه البنية بالشكل التالي:

	0	1	2	3	4	5	
myArray							
	← front							
	back →							

introduction to queues 2

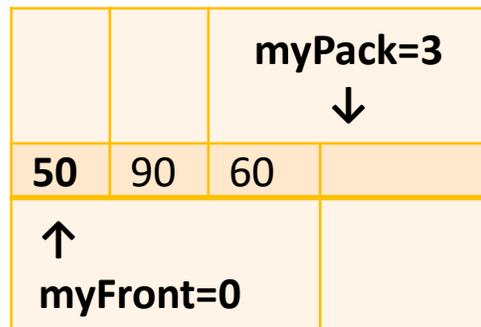
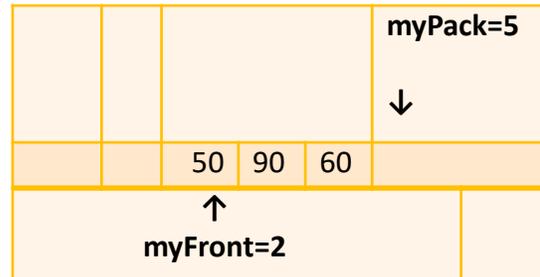


تم إضافة عنصر إلى الرتل من خلال تخزينه في الموقع myBack من المصفوفة ثم تتم زيادة قيمة myPack بمقدار 1, حيث أن قيمة myBack لا تتجاوز قيمة عظمى هي QUEUE_CAPACITY للمصفوفة. إن مصدر الصعوبة في هذا التحقيق للرتل يتمثل في احتمال تتجاوز نهاية المصفوفة, فإن العناصر قد تزاح إلى بداية المصفوفة.

لتوضيح هذه الفكرة, لنفرض أن لدينا رتلاً فيه QUEUE_CAPACITY=5 وعناصره من النوع الصحيح, عندئذ تتالي العمليات التالي: AddQ 70, AddQ 80, AddQ 50 سينتج الوضع التالي في الرتل:

لنفرض الآن أننا أزلنا عنصرين من الرتل:

introduction to queues 2



وبإضافة العنصرين 90 , 60 يصبح الوضع كما يلي:

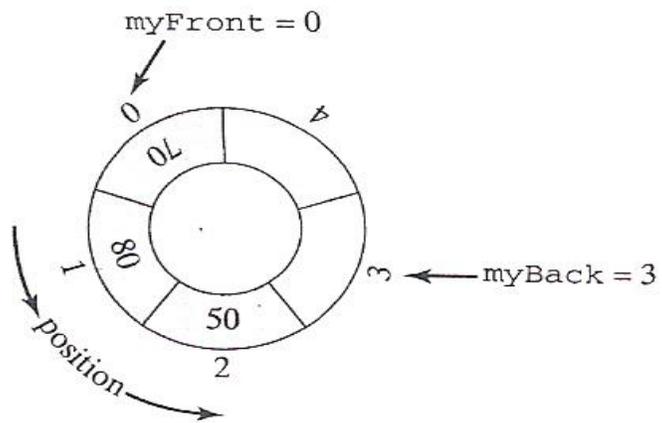
ملاحظة: عند محاولة إضافة عنصر سيخرج دليل النهاية من نطاق حجز المصفوفة. وقبل أن يكون لدينا الإمكانية لإضافة عناصر أخرى إلى الرتل, يجب أن تراح العناصر إلى بداية المصفوفة.

عملية الإزاحة تلك لعناصر المصفوفة غير فعالة على الإطلاق, خاصة عندما تكون العناصر عبارة عن سجلات ضخمة.

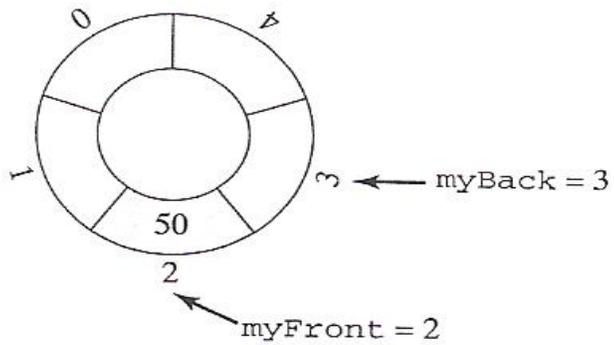
يمكن تجنب تلك العملية بجعل المصفوفة دائرية circular, يكون العنصر الأول يلي العنصر الأخير, وهذا الأمر يمكن تحقيقه بفهرسة المصفوفة بدءاً من الصفر, وزيادة myFront و myPack بإضافة باقي القسمة على QUEUE_CAPACITY.

إذا أردنا تمثيل العمليات السابقة وفق هذا الأسلوب ينتج ما يلي:

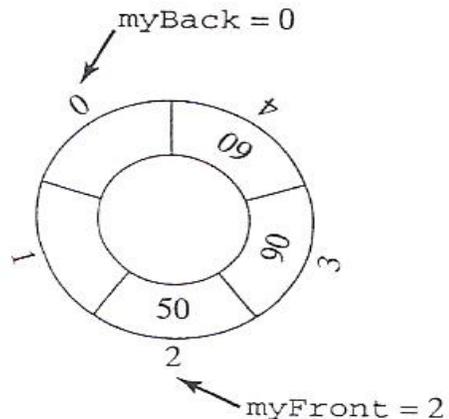
addQ 70
addQ 80
addQ 50



removeQ
removeQ



addQ 90
addQ 60



إن عملية إضافة تالية تصبح الآن متاحة بدون الحاجة لتحريك أي عنصر من مكانه لأن myPack يشير الآن إلى الموقع 0. لنبدأ الآن بتعريف عملية اختبار فيما إذا كان الرتل فارغاً. إذا احتوى الرتل على عنصر واحد، سيكون في الموقع myFront من المصفوفة، وسيكون myPack هو الموقع الشاغر التالي له. إذا تم حذف هذا العنصر، تتم زيادة myFront بمقدار 1 وبالتالي سيكون myFront, myBack لهما نفس القيمة، وبالتالي لتحديد فيما إذا كان الرتل فارغاً يجب أن نختبر الشرط التالي (myFront==myBack).

أما في عملية البناء، فإن باني الرتل سيقوم بتهيئة كلا من قيمتي myFront و myBack إلى الصفر.

introduction to queues 2

إن هذا التحقيق للرتل يطرح إمكانية حصول امتلاء مواقع الرتل, لتبيان هذه الحالة نفرض أن لدينا رتلاً يحتوي على موقع واحد فارغ, إذا قمنا بتخزين عنصر في هذا الموقع فإن قيمة myPack ستزداد بمقدار 1 وبالتالي ستصبح مساوية لقيمة myFront, وهذا هو نفس الشرط الذي يشير إلى حالة كون الرتل فارغاً, أي لا نستطيع في هذه الحالة التفريق بين حالي كون الرتل فارغاً أو ممتلئاً. يمكن تجنب هذه العملية بالإبقاء دائماً على موقع فارغ دائماً ضمن الرتل, وبالتالي يصبح شرط اعتبار الرتل ممتلئاً هو $(myBack+1)\%QUEUE_CAPACITY==myFront$.

هناك العديد من الطرق للإستفادة من الموقع الفارغ في بداية المصفوفة, إحدى الطرق الشائعة هي إضافة عنصر بيانات من النوع الصحيح count إلى الصنف يقوم بتخزين عدد العناصر الموجودة حالياً في الرتل. طريقة أخرى تتمثل في استخدام عنصر بيانات يدل أن الرتل full بدلاً من عنصر البيانات الصحيح count, إذا أصبح الرتل ممتلئاً يأخذ القيمة 0 للدلالة عن false وإلا يأخذ هذا العنصر أبة قيمة للدلالة عن true. نلخص هذا التعريف للصنف بالملف Queue.h المبين فيما يلي:

introduction to queues 2



1- مدخل إلى الأرتال 2

```
// queue2.cpp : main project file.
```

```
#include "stdafx.h"
```

```
#include <iostream>
```

```
using namespace std;
```

```
#ifndef QUEUE
```

```
#define QUEUE
```

```
const int QUEUE_CAPACITY=128;
```

```
typedef int QueueElement;
```

```
class Queue
```

```
{/** function members **/
```

```
public:
```

```
Queue();
```



introduction to queues 2

```

bool empty() const;
QueueElement front() const;
void addQ(const QueueElement & value);
void removeQ(); void display() const;
/**** data members ****/
private: QueueElement myArray[QUEUE_CAPACITY]; int myFront, myBack;
}; //end of class declaration

inline Queue::Queue() { myFront=myBack=0; }
inline bool Queue::empty() const { return (myFront==myBack) ;}
void Queue::addQ(const QueueElement & value)
{ int newBack=(myBack+1)%QUEUE_CAPACITY;
  if (newBack==myFront)cout<<"**** QUEUE IS FULL ****"<<endl;
  else{ myArray[myBack]=value; myBack=newBack; }}
QueueElement Queue::front() const
{ if (!empty()) return myArray[myFront]; cerr<<"**** QUEUE IS EMPTY ***\n"; }

```

introduction to queues 2

```
void Queue::removeQ() { if (empty()) cout<<"*** QUEUE IS EMPTY ***\n";
                        else myFront=(myFront+1)%QUEUE_CAPACITY;}
void Queue::display() const { for (int i=myFront ; i<myBack ; i++) cout<<myArray[i]<<" "; }
#endif

void main() { Queue q1; cout<<"Queue is empty "<<q1.empty( )<<endl;
             q1.addQ(7); q1.addQ(77);q1.addQ(5); q1.addQ(55); cout<<"Queue is empty "<<q1.empty( )<<endl;
             q1.display();
             q1.removeQ();q1.removeQ();cout<<"\n\n";q1.display(); cout<<"\n\n";system("pause");}
```

Queue is empty 1

Queue is empty 0

7 77 5 55

5 55

Press any key to continue . . .

applications of queues 1

مسألة التمارين الحسابية: بفرض أننا نريد تصميم برنامج لإنجاز تمارين على العمليات الحسابية الأساسية, وبشكل أكثر تحديداً, تمارين على إنجاز عملية الجمع لأعداد صحيحة مولدة بشكل عشوائي. إذا قام الطالب بالإجابة بشكل صحيح يتم توليد مسألة جديدة, ولكن إذا أجاب بشكل خاطئ فإن المسألة يتم تخزينها بحيث يتم إعادة طرحها لاحقاً في نهاية الجلسة.

الأغراض والعمليات في المسألة: إن أحد الأغراض في هذه المسألة هو مسألة الجمع وبالتالي يمكن توصيفها على شكل صنف يدعى AdditionProblem له الصفات التالية:

البيانات الأعضاء: الأعداد المراد جمعها (addends) أعداد صحيحة عشوائية .
النتائج answer عدد صحيح.

العمليات: الحصول على مسألة عشوائية بعددين ضمن مجال معين.
عرض المسألة.

إيجاد جواب المسألة.

اختبار الجواب المعطى فيما إذا كان صحيحاً.

أما باقي الأغراض في المسألة فهي:

1 applications of queues

numProblem (int) : عدد المسائل المراد طرحها.

maxAddend (int) : أكبر عدد ممكن استخدامه ضمن مسألة الجمع.

userAnswer (int) : إجابة المستخدم عن المسألة.

count (int) : عدد المسائل المطروحة.

wrong (int) : عدد المسائل التي تمت الإجابة عليها بشكل خاطئ للمرة الثانية.

cout (ostream) : مجرى لإخراج النتائج, المسائل, الإحصائيات.

cin (istream) : مجرى إدخال المسائل, الإجابات, وباقي المعلومات.

لإنجاز حل هذه المسألة سنستخدم غرضاً من الصنف queue لتخزين المسائل التي لم تتم الإجابة عليها بشكل صحيح يدعى wrongQueue بحيث يتم تخزين المسائل التي تتلقى إجابات خاطئة واحدة إثر الأخرى ومن ثم يتم إخراجها من هذا الرتل في نهاية الجلسة والإجابة عليها واحدة تلو الأخرى.

كما أننا نحتاج إلى آلية لتوليد الأعداد العشوائية, ويتم ذلك من خلال التابع (rand()) الموجود ضمن المكتبة القياسية. من أجل الحصول على نمذجة أفضل للحل سنقوم بتعريف نمط بيانات يدعى RandomInt كصنف من خلال الملف الرأسي RandomInt.h التالي:

applications of queues 1

```

#ifdef RANDOMINT
#define RANDOMINT
#include <iostream>
#include <cstdlib>
#include <cassert>
#include <ctime>
using namespace std;
class RandomInt
{ public:      RandomInt();      RandomInt(int low, int high);
  RandomInt(int seedValue);      RandomInt(int seedValue, int low, int high);
  void Print(ostream & out) const;      RandomInt Generate();
  RandomInt Generate(int low, int high);      operator int();
private: void Initialize(int low, int high);
  void SeededInitialize(int seedValue, int low, int high);
  int NextRandomInt();
  int      myLowerBound,      myUpperBound,      myRandomValue;
  static bool generatorInitialized;
};

```

applications of queues 1

```
inline RandomInt::RandomInt(){ Initialize(0, RAND_MAX);}
inline RandomInt::RandomInt(int low, int high)
{ assert(0 <= low); assert(low < high); Initialize(low, high);}
inline RandomInt::RandomInt(int seedValue)
{ assert(seedValue >= 0); SeededInitialize(seedValue, 0, RAND_MAX);}
inline RandomInt::RandomInt(int seedValue, int low, int high)
{ assert(seedValue >= 0); assert(0 <= low); assert(low < high);
  SeededInitialize(seedValue, low, high);}
inline void RandomInt::Print(ostream & out) const
{ out << myRandomValue;}
inline ostream & operator<< (ostream & out, const RandomInt & randInt)
{ randInt.Print(out); return out;}
inline int RandomInt::NextRandomInt()
{ return myLowerBound + (rand() % (myUpperBound - myLowerBound + 1));}
inline RandomInt RandomInt::Generate()
{ myRandomValue = NextRandomInt(); return *this;}
inline RandomInt::operator int(){ return myRandomValue;}
#endif
```

applications of queues 1

يمكن تعريف التوابع الأعضاء في الصنف من خلال الملف RandomInt.cpp كما يلي:

```
#include "stdafx.h"
#include "RandomInt.h"
using namespace std;
bool RandomInt::generatorInitialized = false;
void RandomInt::Initialize(int low, int high)
{ myLowerBound = low; myUpperBound = high;
  if (!generatorInitialized)
    { srand(long(time(0))); generatorInitialized = true; }
  myRandomValue = NextRandomInt();
}
void RandomInt::SeededInitialize(int seedValue, int low, int high)
{ myLowerBound = low; myUpperBound = high; srand(seedValue);
  generatorInitialized = true; myRandomValue = NextRandomInt();
}
RandomInt RandomInt::Generate(int low, int high)
{ assert(0 <= low); assert(low < high); myLowerBound = low; myUpperBound = high;
  myRandomValue = NextRandomInt(); return *this;}
```

applications of queues 1

3- تطبيقات على الأرتال 1

يمكن أخيراً وضع خوارزمية حل مسألة التمارين الحسابية كما يلي:

1. قراءة numProblems و maxAddend.
2. من أجل count ضمن المجال 1 إلى numProblems قم بما يلي:
 - a. قم بتوليد وعرض مسألة.
 - b. اقرأ userAnswer.
 - c. إذا كان userAnswer صحيح قم بعرض رسالة تفيد بأن الإجابة صحيحة وإلا قم بعرض رسالة تفيد بأن الإجابة غير صحيحة وأضف المسألة إلى wrongQueue.
 3. قم بعرض التعليمات واجعل wrong=0.
 4. طالما أن wrongQueue غير فارغ, قم بما يلي:
 - a. اجلب واحذف مسألة من بداية الرتل.
 - b. اعرض المسألة.
 - c. اقرأ userAnswer.
 - d. إذا كان userAnswer صحيح قم بعرض رسالة تفيد بأن الإجابة صحيحة وإلا قم بعرض رسالة تفيد بأن الإجابة غير صحيحة وقم بزيادة wrong بمقدار 1.
 5. اعرض قيمة wrong.

بالتالي يصبح حل مسألة التمارين الحسابية كما يلي:
الملف AdditionProblem.h:

applications of queues 1

```

#include "RandomInt.h"
#include <iostream>
using namespace std;
#ifndef ADDITION_PROBLEM
#define ADDITION_PROBLEM
using namespace std;
class AdditionProblem
{
    /**** Function Members ****/
public:
    void Get(int maxAddend)
    { myAddend1.Generate(0, maxAddend); myAddend2.Generate(0, maxAddend); }
    void Display(ostream & out)
    { out << myAddend1 << " + " << myAddend2 << " = ? "; }
    int Answer(){ return (myAddend1 + myAddend2); }
    bool Correct(int userAnswer) { return (userAnswer == Answer()); }
    /**** Data Members ****/
private:
    RandomInt myAddend1, myAddend2;
};
#endif

```

applications of queues 1



3- تطبيقات على الأرتال 1

الملف driver.cpp للاختبار:

```
// queue4.cpp : main project file.
#include "stdafx.h"
#include "AdditionProblem.h"
#include <queue>
#include <iostream>
using namespace std;
void main()
{   int numProblems,           maxAddend;
    cout << "How many problems would you like? ";           cin >> numProblems;
    cout << "What's the largest addend you would like? ";   cin >> maxAddend;

    queue<AdditionProblem>     wrongQueue;
    AdditionProblem problem;   int userAnswer;
    for (int count = 1; count <= numProblems; count++)
    {   problem.Get(maxAddend);           problem.Display(cout);           cin >> userAnswer;
        if (problem.Correct(userAnswer))           cout << "Correct!\n\n";
        else {   cout << "Sorry -- Try again later\n\n";           wrongQueue.push(problem);           }
    }
}
```

applications of queues 1

```
cout << "\nIf you got any problems wrong, you will now be given"
        "\na second chance to answer them correctly.\n";

int wrong = 0;
while (!wrongQueue.empty())
{
    problem = wrongQueue.front();    wrongQueue.pop();    problem.Display(cout);
    cin >> userAnswer;
    if (problem.Correct(userAnswer))    cout << "Correct!\n\n";
    else
    {
        cout << "Sorry -- correct answer is " << problem.Answer() << "\n\n";
        wrong++;
    }
}
cout << "\nYou answered " << wrong << " problem"
        << (wrong > 1 ? "s " : " ") << "incorrectly\n";
system("pause");
}
```



How many problems would you like? 4

What's the largest addend you would like? 199

$$175 + 183 = ? 200$$

Sorry -- Try again later

$$76 + 108 = ? 184$$

Correct!

$$117 + 13 = ? 130$$

Correct!

$$142 + 107 = ? 300$$

Sorry -- Try again later

If you got any problems wrong, you will now be given a second chance to answer them correctly.

$$175 + 183 = ? 358$$

Correct!

$$142 + 107 = ? 200$$

Sorry -- correct answer is 249

You answered 1 problem incorrectly

Press any key to continue . . .



Lists	اللوائح 1
	1- مقدمة اللوائح Lists
	1- تحقيق اللوائح باستخدام التخزين المتتالي
	2- مدخل إلى اللوائح المترابطة
	3- تحقيق اللوائح المترابطة باستخدام المصفوفات
	4- المؤشرات في C++
	5- تخصيص وإلغاء تخصيص الذاكرة وقت التنفيذ
	6- تحقيق اللوائح المترابطة في لغة C++ باستخدام المؤشرات
	7- قالب الصنف list القياسي

References

- Deitel & Deitel, Java How to Program, Pearson; 10th Ed(2015)

- د.علي سليمان، بني معطيات بلغة JAVA، بني معطيات بلغة C++، بني معطيات بلغة Pascal جامعة تشرين 2014، 2007، 1998

1- Introduction

1- مقدمة:

- إن المكذسات، والأرتال ثنائية الطرف التي قمنا بدراستها في الفصول السابقة هي أنواع خاصة من اللوائح،
- كل من هذه الأنماط المجردة للبيانات هو عبارة عن تتالي من عناصر البيانات والعمليات الأساسية عليها درسنا منها الإضافة والحذف.
- إن عمليات الإضافة والحذف في هذه البنى مقيدة بنهايات اللوائح، في حين لا توجد مثل هذه القيود على اللوائح العامة حيث يمكن إضافة العناصر أو حذفها في أي مكان من اللائحة.
- ندرس في هذا الفصل بشكل أكثر تفصيلاً هذه اللوائح العامة وتحقيقاتها المختلفة.

1- تحقيق اللوائح باستخدام التخزين المتتالي 1

يعد استخدام اللوائح بأشكالها المختلفة أمراً شائعاً في حياتنا اليومية، فهناك لوائح المشتريات، لوائح الديون، لوائح الصف، لوائح الموظفين، لوائح البريد.. وغيرها، أو حتى لوائح اللوائح. وجميع هذه الأشكال من اللوائح تحمل صفات مشتركة ويمكن من خلالها وضع التعريف التالي للائحة:

اللائحة كنمط بيانات مجرد

مجموعة من عناصر البيانات:

تتال منته (مجموعة مرتبة) من عناصر البيانات.

العمليات الأساسية:

- ◀ البناء construction: إنشاء لائحة فارغة.
- ◀ اختبار كون اللائحة فارغة empty.
- ◀ التجول traverse: التجول عبر اللائحة أو عبر جزء منها،
- ◀ البحث search الوصول إلى العناصر ومعالجتها بالترتيب.
- ◀ الإدراج insert: إضافة عنصر في أي مكان من اللائحة.
- ◀ الحذف delete: حذف عنصر من أي مكان من اللائحة.

البنية التخزينية:

إن تخزين عناصر اللائحة ضمن مصفوفة أمر غير مناسب نظراً لعيوب المصفوفات من حجم ثابت وأهم العيوب هي التعديل عليها من إضافة وحذف.

23	25	34	48	61	79	82	89	91	99	?	...	?
----	----	----	----	----	----	----	----	----	----	---	-----	---

23	25	34	48	56	61	79	82	89	91	99	...	?
----	----	----	----	----	----	----	----	----	----	----	-----	---

23	25	34	48	56	61	79	82	89	91	99	...	?
----	----	----	----	----	----	----	----	----	----	----	-----	---

23	34	48	56	61	79	82	89	91	99	99	...	?
----	----	----	----	----	----	----	----	----	----	----	-----	---

نريد القيمة 56 بعد العنصر 48 في لائحة الأعداد الصحيحة:

23,25,34,48,61,79,82,89,91,99

بحيث تصبح اللائحة الجديدة

23,25,34,48,56,61,79,82,89,91,99

عند الرغبة بحذف العنصر 25 سنجد العملية المعاكسة وهي الإزاحة نحو اليسار لتأتي هذا العنصر.

نظراً لهذه العيوب تطلب الأمر البحث عن بنية أكثر فعالية.

1 introduction to linked lists

اللائحة هي تتالي من عناصر البيانات، وهذا يعني أن هناك ترتيباً مرتباً بالعناصر في اللائحة: فهي تحوي عنصر أول، عنصر ثاني... وهكذا، وبالتالي فإن أي تحقيق لنمط البيانات المجرد هذا يجب أن يحتوي طريقة لتحديد هذا الترتيب.

عملية الترتيب هذه لعناصر اللائحة محققة ضمناً **implicitly** من خلال الترتيب الطبيعي لعناصر المصفوفة، حيث العنصر الأول مخزن في الموقع الأول في المصفوفة، العنصر الثاني مخزن في الموقع الثاني للمصفوفة، وهكذا. إن هذا التوصيف الضمني للترتيب لعناصر اللائحة هو الذي يتطلب إزاحتها في المصفوفة عند حشر العناصر أو حذفها مسبقاً عدم فعالية التحقيق للوائح الديناميكية (أي التي تتغير بكثرة بسبب عمليات الحشر والحذف). نتعرف في هذه الفقرة على طريقة أخرى لتحقيق اللوائح تتخلص من هذا العيب من خلال التحديد الصريح **explicitly** لترتيب عناصر اللائحة.

ما هي عمليات اللوائح المترابطة:

في أي بنية مستخدمة لتخزين عناصر لائحة ما، و المحافظة على ترتيب عناصر اللائحة، يجب أن يتحقق على الأقل أداء العمليات التالية:

- ◀ تحديد موقع العنصر الأول.
- ◀ بمعرفة موقع أي عنصر في اللائحة،
- ◀ إيجاد العنصر التالي.
- ◀ تحديد موقع نهاية اللائحة.

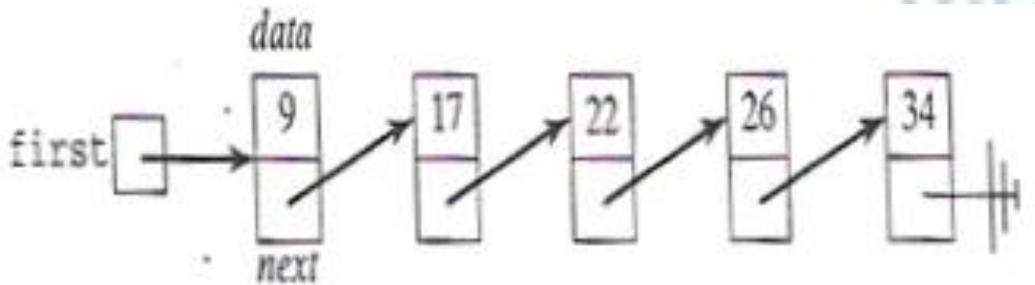
introduction to linked lists 2

إن الرغبة في تحسين الفعالية والتبسيط لهذه العملية هو من قاد إلى اللوائح المترابطة.

اللائحة المترابطة linked list هي مجموعة مرتبة من العناصر تدعى العقد nodes كل منها تتكون من جزئين على الاقل:

1. جزء بيانات data part لتخزين عنصر من اللائحة.
2. جزء التالي next part لتخزين رابط link أو مؤشر pointer يشير إلى موقع العقدة الحاوية للعنصر التالي في اللائحة، إذا لم يكن هناك عنصر تالي، عندها تستخدم القيمة الخاصة null value.

على الرغم من أن موقع العقدة التي تحوي العنصر الأول يجب أن يكون محددًا، فإنه سيكون القيمة null إذا كانت اللائحة فارغة. للتوضيح، اللائحة المترابطة التي تحتوي القيم 9,17,22,26,34 يمكن تمثيلها كما في الشكل التالي:



في هذا الشكل، تمثل الأسهم الروابط، المؤشر first يشير إلى العقدة الأولى في اللائحة. جزء البيانات في كل عقدة يخزن عدداً صحيحاً من اللائحة، ورمز الأرضي ground symbol في العقدة الأخيرة يمثل رابطاً فارغاً للإشارة إلى أن عنصر اللائحة هذا ليس له تالي، وتوضع القيمة null للدلالة على النهاية.

introduction to linked lists 3

تحقيق العمليات الأساسية للائحة:

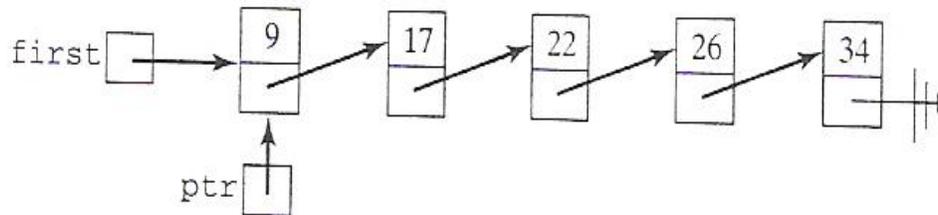
سنبين فيما يلي كيفية تحقيق العمليات الأساسية المبينة في الفقرة السابقة على اللوائح المترابطة:
عملية البناء construction: لبناء لائحة فارغة، نستطيع ببساطة جعل المؤشر first يشير إلى القيمة null للإشارة إلى أنها لا تشير إلى أي عقدة:

first=null_value;



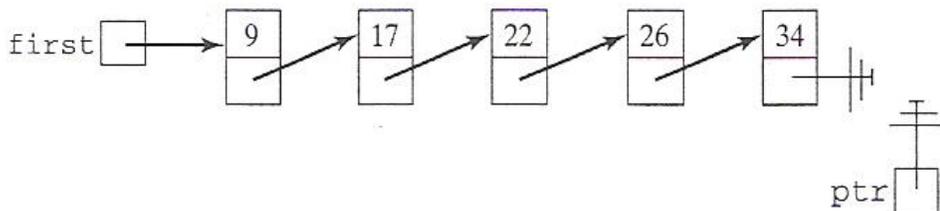
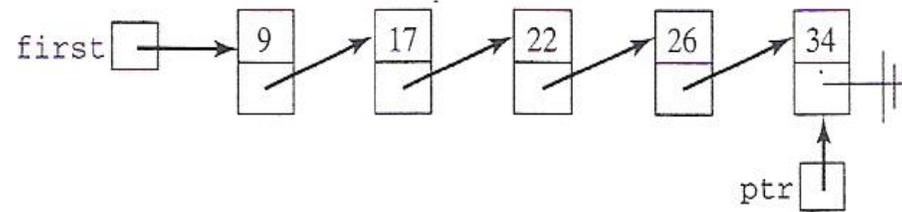
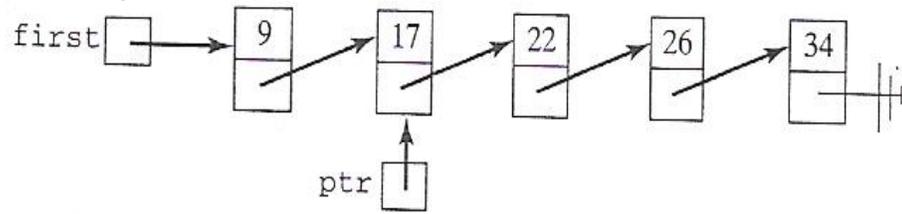
العملية empty: يمكن ببساطة تنفيذ العملية الثانية من عمليات اللائحة وهي تحديد فيما إذا كانت اللائحة فارغة من خلال اختبار كون first يشير إلى null كما يلي:
first==null_value ?

العملية traverse: العملية الأساسية الثالثة هي التجول عبر اللائحة. للتجول عبر لائحة مترابطة (كلائحة الأعداد الصحيحة المشار إليها سابقاً) نبدأ بتهيئة مؤشر مساعد ptr ليشير إلى العقدة الأولى ونعالج قيمة عنصر اللائحة 9 المخزنة في هذه العقدة.



3- مدخل إلى اللوائح المترابطة 4

introduction to linked lists 4



لانتقال إلى العقدة التالية، نتبع الرابط من العقدة الحالية وذلك بجعل ptr مساوي للرابط في العقدة التي يشير إليها ptr (هذه العملية شبيهة بزيادة الدليل بمقدار 1 في التحقيق باستخدام التخزين المتتالي) ومن ثم معالجة العدد الصحيح 17 المخزن في تلك العقدة.

نستمر في هذه العملية إلى أن نصل إلى العقدة الحاوية على القيمة 34:

إذا حاولنا الانتقال إلى العقدة التالية، سيصبح ptr مشيراً إلى null معبراً عن نهاية اللائحة.

introduction to linked lists 5

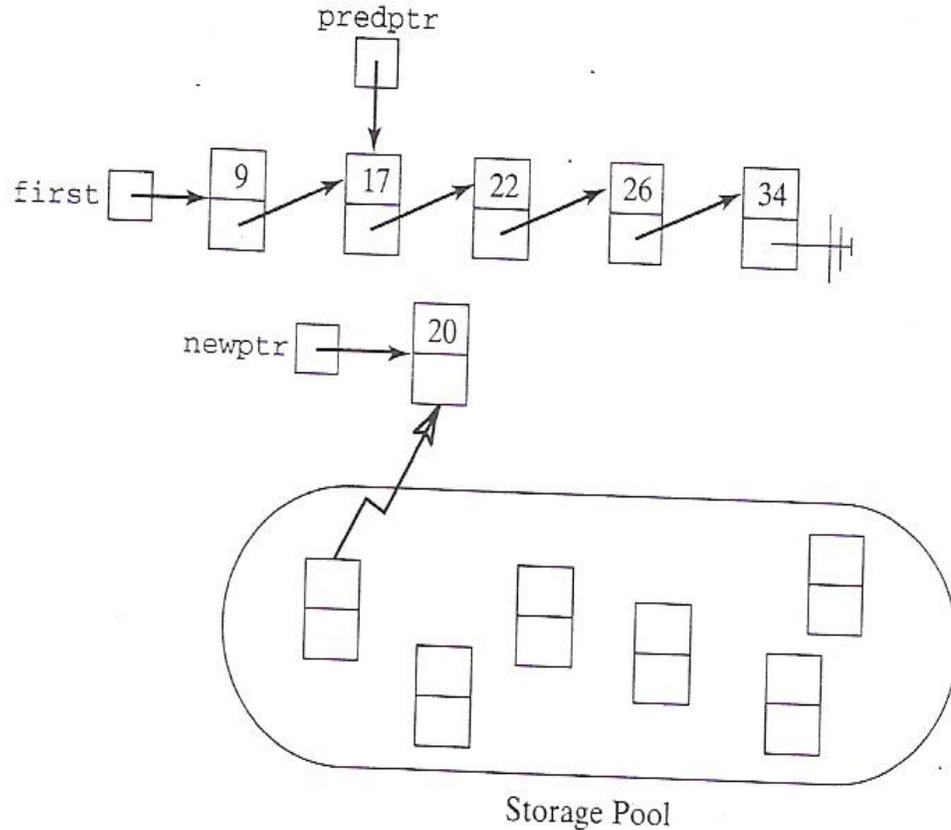
وكخلاصة، يمكن التجول عبر اللائحة المترابطة كما يلي:

```
ptr=first;
while(ptr!=null_value)
{
    process data part of node pointed to by ptr.
    ptr= next part of node pointed to by ptr.
}
```

إن هذه الخوارزمية صحيحة حتى من أجل اللائحة الفارغة، حيث تشير في هذه الحالة first إلى القيمة null ويتم تجاوز الحلقة. لعرض محتويات اللائحة فإن المعالجة في هذه الحالة هي ببساطة إخراج جزء البيانات من العقدة. أما للبحث عن قيمة معطاة item فستكون المعالجة في هذه الحالة:

```
if (item==data part of node pointed to by ptr)
    terminate the loop //ptr points to the node containing item
```

introduction to linked lists 6

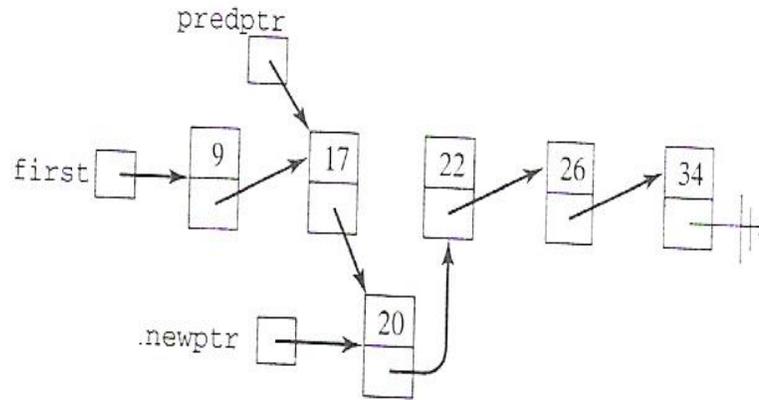
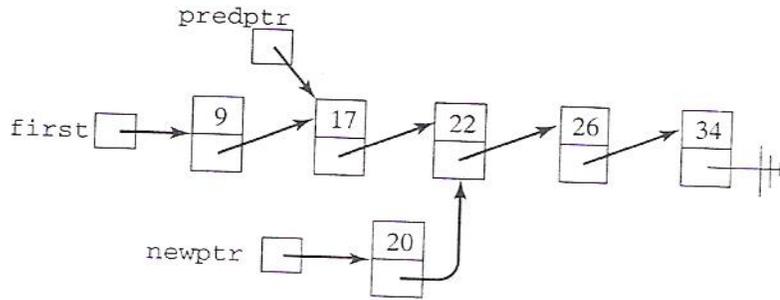


العملية insert: لحشر قيمة بيانات جديدة إلى لائحة مترابطة، يجب أولاً إنشاء عقدة جديدة وتخزين هذه القيمة في جزء البيانات الخاص بها. سنفرض أن هناك عملية ما يمكن استخدامها لإنشاء عقدة من مجمع التخزين storage pool. الخطوة التالية هي ربط هذه العقدة الجديدة باللائحة الموجودة، للقيام بذلك هناك حالتان يجب الانتباه إليهما: الأولى الحشر بعد عنصر ما ضمن اللائحة، والثانية هي الحشر في بداية اللائحة.

لتوضيح الحالة الأولى، لنفرض أننا نريد حشر 20 بعد 17 في اللائحة السابقة، وبفرض أن المؤشر predptr يشير إلى العقدة الحاوية على القيمة 17. بداية علينا الحصول على عقدة جديدة ندعوها newptr ونقوم بتخزين 20 في جزء البيانات الخاص بها.

3- مدخل إلى اللوائح المترابطة 7

introduction to linked lists 7

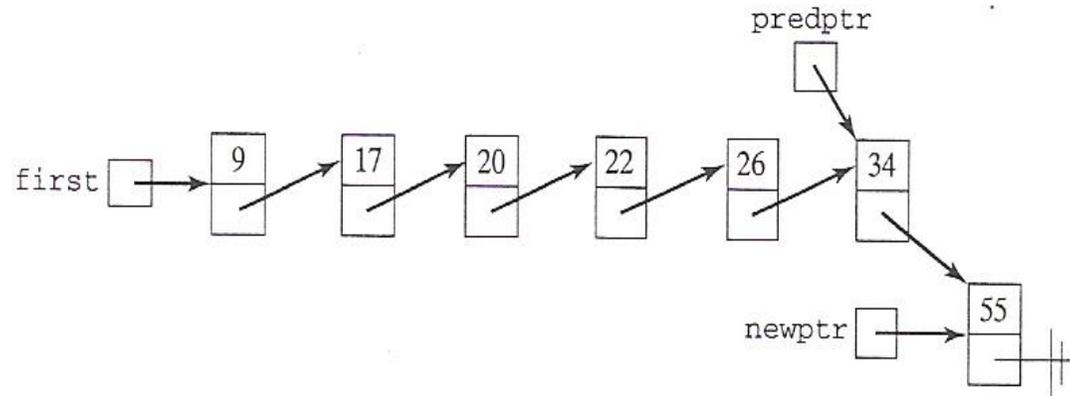


نقوم بحشرها في اللائحة بجعل الجزء `next` الخاص بها يساوي الجزء التالي للعقدة المشار إليها بالمؤشر `predptr`

حالياً نجعل الجزء `next` للعقدة المشار إليها بالمؤشر `predptr` يشير إلى العقدة الجديدة.

لاحظ أن عملية الحشر هذه تعمل في حالة الحشر إلى نهاية اللائحة أيضاً.

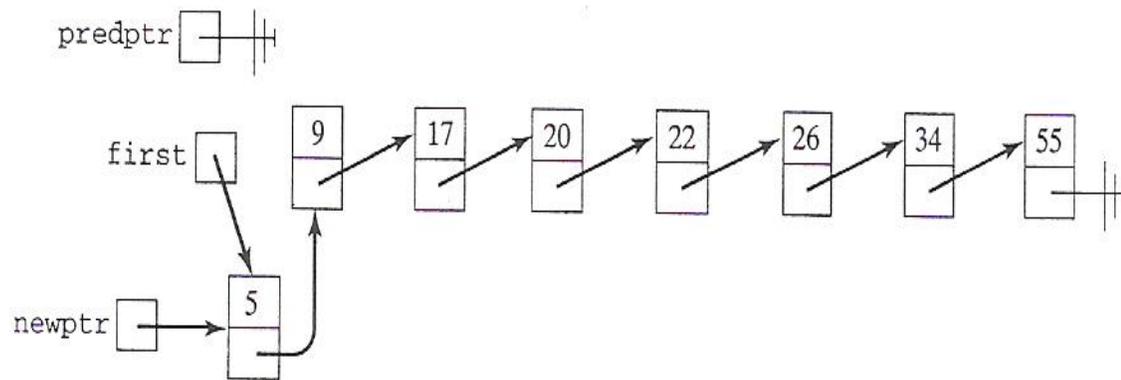
introduction to linked lists 8



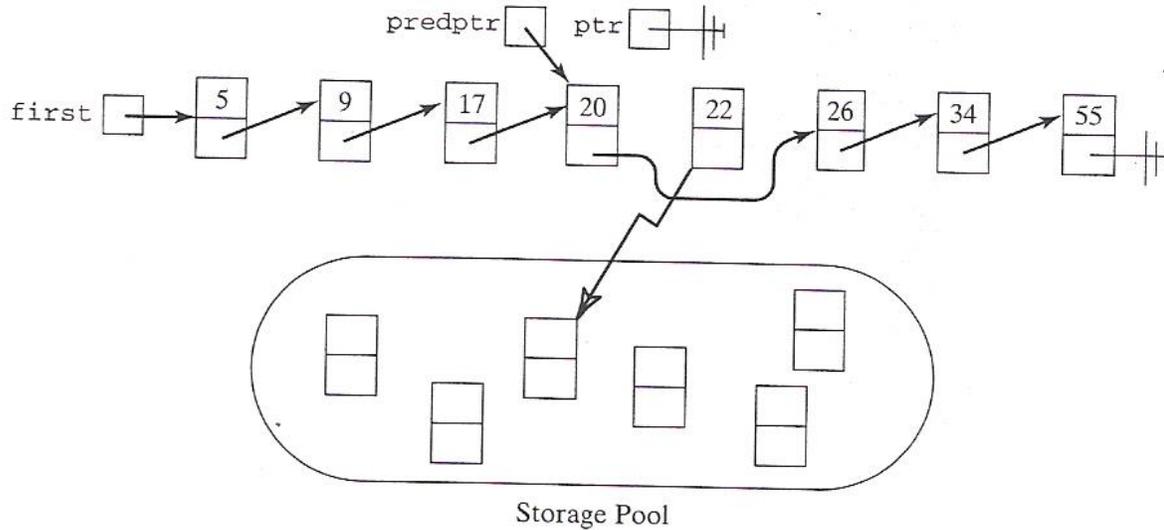
الشكل التالي يوضح عملية إضافة القيمة 55 إلى نهاية اللائحة.

الحالة الثانية الحشر في بداية اللائحة لنفرض أننا نريد حشر عقدة تحوي القيمة 5.

لتوضيح الحالة الثانية، الخطوتان الأولى والثانية هما نفسهما كما في الحالة الأولى مع العلم أن 'predptr' هي 'first' أما الخطوة الثالثة فتمثل جعل المؤشر 'first' يشير إلى العقدة الجديدة.



introduction to linked lists 9



العملية delete: للحذف هناك حالتان نميزهما:

الأولى حذف عنصر له عنصر سابق

الثانية حذف العنصر الأول في اللائحة. سنفرض أن

هناك عملية ما يمكن استخدامها لإعادة عقدة مشار

إليها بمؤشر محدد إلى مجمع التخزين storage pool.

الحالة الأولى عنصر له عنصر سابق:

مثال: حذف العقدة الحاوية على القيمة 22 من اللائحة،

يشير المؤشر ptr إلى العقدة المراد حذفها، والمؤشر

predptr يشير إلى العقدة السابقة لها (العقدة الحاوية

على 20).

يمكن القيام بعملية الحذف بجعل الرابط في العنصر

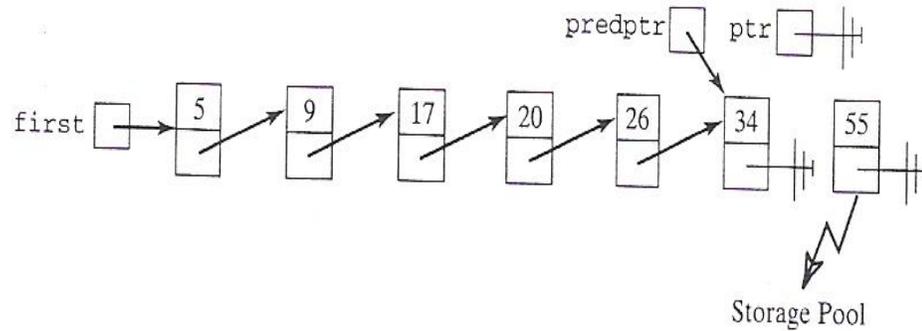
السابق يشير إلى العنصر اللاحق للعقدة المراد حذفها.

ومن ثم إعادة العقدة المشار إليها بالمؤشر ptr إلى مجمع

التخزين للعقد غير المحجوزة.

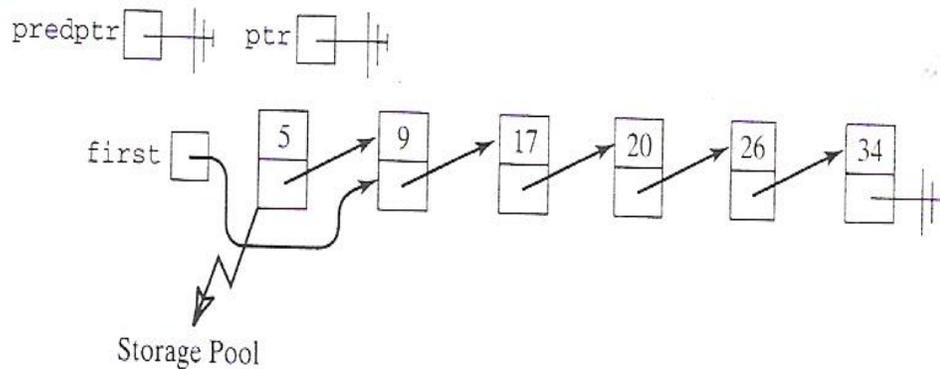
3- مدخل إلى اللوائح المترابطة 9

introduction to linked lists 9



لاحظ أن هذه الخوارزمية تعمل في حال الحذف من نهاية اللائحة أيضاً.

الحالة الثانية حذف العقدة الأولى:



سهلة وتقتصر على جعل المؤشر first يشير إلى العقدة الثانية في اللائحة.

ومن ثم إعادة العقدة المحذوفة من اللائحة إلى مجمع التخزين.

introduction to linked lists 9

إن العرض السابق يبين أنه من الممكن حشر عنصر في لائحة مترابطة في موقع محدد أو حذف عنصر من موقع محدد بدون إزاحة عناصر اللائحة. إن هذا يعني أنه بخلاف التحقيق باستخدام التخزين المتتالي فإن هذه العمليات تتم خلال مدة ثابتة ودرجة تعقيدها $O(1)$.

تعرفنا حتى هذه اللحظة على اللوائح المترابطة بشكل مجرد فقط ولكننا لم ندرس تحقيقها. لتحقيق اللوائح المترابطة يجب أن نمتلك على الأقل المقدرات التالية:

1. بعض الوسائل لتقسيم الذاكرة إلى عقد، كل منها تتألف من جزء بيانات وجزء رابط، وبعض التحقيقات للمؤشرات.
2. عمليات للوصول إلى قيم مخزنة في كل عقدة، أي عمليات للوصول إلى جزء البيانات وجزء التالي من العقدة المشار إليها بمؤشر ما.
3. بعض الوسائل للإمساك بالعقد التي هي قيد الاستخدام بالإضافة إلى العقد الحرة ولتبادل العقد بين تلك التي هي قيد الاستخدام ومجمع العقد الحرة.

introduction to linked lists 9

بنية العقدة:

نذكر بأن العقدة في لائحة مترابطة تتألف من جزئين أو أكثر: جزء أو أجزاء بيانات data part يخزن عنصر من اللائحة وجزء التالي next part يشير إلى العقدة الحاوية على القيمة التالية لهذه القيمة أو على القيمة null إذا كانت هي آخر عقده في اللائحة. إن هذا يفترض بأن كل عقدة يمكن تمثيلها كسجل struct أو صنف class واللائحة المترابطة هي مصفوفة من السجلات. كل سجل في هذه الحالة سيتضمن عضوين: عضو بيانات يخزن قيمة عنصر اللائحة والعضو التالي الذي يشير إلى التالي من خلال تخزين دليله في المصفوفة. وبالتالي يمكن التصريح عن اللائحة المترابطة كبنية تخزينية باستخدام المصفوفات كما يلي:

```
/** Node declaration */
struct NodeType
{DataType data; int next;};
const int NULL_VALUE=-1; //a nonexistent location
/** The Storage Pool */
const int NUMNODES=2048;      NodeType node[NUMNODES];
int free;          // points to a free node
```



انتهت المحاضرة الخامسة