DATA STRUCTURES AND ALGORITHMS

اهداف المقرر :

1- تعليم الطالب كيفية تمثيل البيانات في ذاكرة الحاسوب
2- تعليم الطالب طرق حجز مواقع الذاكرة الديناميكية والثابتة static and dynamic
3- تعليم الطالب هياكل البيانات الغير خطية (Tree and Graph) وتركيبها وبرمجتها
4- التعرف على اهم تطبيقات هياكل البيانات في الحاسبة
5- تعليم الطالب طرق و تقنيات البحث والترتيب للبيانات في كل هيكلية من هياكل البيانات

اللغة المستخدمة فب الجانب العملي :

C++

المصادر العلمية للمقرر:

1- هياكل البيانات بلغة C++ للمؤلف عصام الصفار
2- Drozdek A. Data Structures and algorithms in C++. Cengage Learning; 2012 Aug 27.
3- Weiss MA. Data structures & algorithm analysis in C++. Pearson Education; 2012 Feb 1.

المقدمة :

تلعب عملية تمثيل البيانات في ذاكرة الحاسوب بصورة كفوءة دورا مهما في في الحصول على النتائج المطلوبة والمساهمة في رفع ادء البرامج المخصصة للتعامل مع البيانات. لذلك يمكن تمثيل البيانات في ذاكرة الحاسوب باستخدام هياكل بيانية مختلفة لكل منها مايميزه. انا اختيار الهيكل البياني المناسب لتمثيل البيانات يخضع لعامل المساحة الخزنية والسرعه في التعامل مع البيانات المخزونه داخل الهيكل البياني. لذلك يمكن تعريف هياكل البيانات (Data structures) على انه دراسة طرق جمع وتمثيل البيانات في ذاكرة الحاسوب عن طريق بناء هياكل بيانية تدعم نوع البيانات المراد خزنها بحيث تمكن المستخدم من الوصل لهذه البيانات والتعامل معها بكفاءة عند اجراء عمليات مثل الاضافة , الحذف و التحديث .... الخ. من ناحية اخرى , عند دراسة هياكل البيانات يجب الاخذ بنظر الاعتبار عاملين مهمين وهما المساحة الخزنية (memory space) والسرعة او عامل الزمن (time) في التعامل مع البيانات. لذلك يجب اختيار الهيكل البياني الذي يقلل من الهدر في ذاكرة الحاسوب وكتابة الخوارزميات الكفوءة التي توفر السرعة عند التعامل مع البينات المخزونة.

تقسم هياكل البيانات الى قسمين وهما الهيكل الفيزياوي (Physical structure) والمنطقي (Logical structure). يقصد بالهيكل الفيزياوي عملية تمثيل البيانات في ذاكرة الحاسوب (Memory). اما الهيكل المنطقي فيمثل نظرة المبرمج لشكل البيانات المخزونة داخل الهيكل البياني. تستخدم لغات البرمجة لبناء عملية الربط بين الهيكل الفيزياوي والمنطقي.

انواع هياكل البيانات :

توفر لغات البرمجة الصيغ المناسبة لتعريف واستخدام العناصر البيانية ذات القيمة الواحدة (المنفردة) فمثلا في لغة C++ تستخدم التعريفات.

```
Int     x  ;
Float   y  ;
Char    A  ;
.       .
```

لتمثل في ذاكرة الحاسوب ويتم التعامل معها بصيغ برمجية بسيطة مثلا :

```
X = 100;
Y =2.3;
A = 'B';
```

اما بالنسبة للعناصر البيانية التي تتكون من عدة قيم بيانية فانها تحتاج لاستخدام هياكل بياني . فيما يلي مخطط يوضح تصنيف هياكل البيانات بشكل عا (Data structures classification):

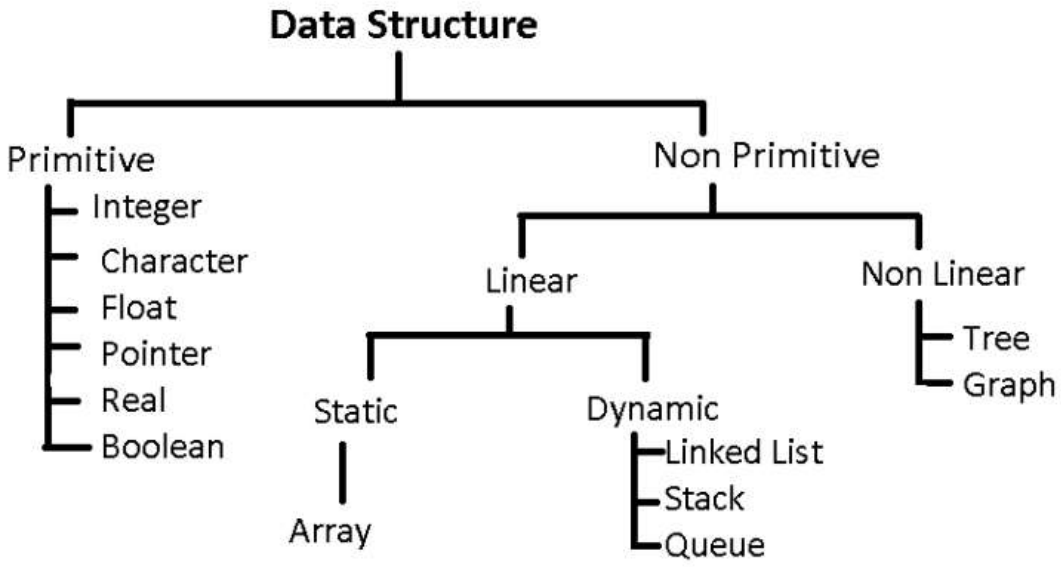


Fig 1: Data structures classification

كيفية أختيار الهيكل البياني المناسب :

1. لكل مجموعة من البيانات هنالك اكثر من طريقة لتنظيمها ووضعها في هيكل بياني معين ويتحدد ذلك وفق عدد من العوامل والاعتبارات لاختيار الهيكل البياني المناسب وهي :
2. حجم البيانات
3. سرعة وطريقة استخدام البيانات
4. السعة الخزنية المطلوبة
5. الزمن اللازم لاسترجاع اية معلومة من الهيكل البياني
6. اسلوب البرمجة

DATA STRUCTURES AND ALGORITHMS

القسم الاول

هياكل البيانات الخطية (Linear structures) :

هو نوع هياكل البيانات الذي يعتمد مبدا التسلسل في خزن عناصر الهيكل البياني في الذاكرة اي بمعنى ان العنصر الاول يتبعه العنصر الثاني والثاني يتبعه العنصر الثالث وهكذا بقية العناصر. تقسم هياكل البيانات الخطية الى قسمين: الاول هو هياكل البيانات الخطية الثابتة (**Static linear structures**) والثاني هو هياكل البيانات الخطية المتحركة (**Dynamic linear structures**) .

- **هياكل البيانات الخطية الثابتة (Static linear structures)**: هي هياكل البيانات التي تستخدم مبدأ تسلسل العناصر وحجز عناوين ذاكرة متجاورة لتمثيل العناصر داخل الهيكل البياني. مثال على ذلك المصفوفة .

المصفوفة Array : .هي عبارة عن مجموعة من المواقع الخزنية في الذاكرة تتصف بما يلي :

1- جميع المواقع تكون من نوع بياني واحد ،حسب صيغة التعريف .....char, int, float
2- يمكن الوصول عشوائيا (Randomly accessed) الى اي موقع من مواقع المصفوفة دون الاعتماد على اي موقع اخر في نفس المصفوفة
3- مقدار الوقت المطلوب للوصول الى اي موقع في المصفوفة هو مقدار ثابت ومتساوي لجميع العناصر .
4- مواقع عناصر المصفوفة تبقى ثابتة ولاتتغير اثناء التعامل مع اي من عناصر المصفوفة.
5- تمثل المصفوفة باستخدام مواقع متعاقبة في الذاكرة .

1- تمثيل المصفوفة الاحادية في الذاكرة 1D array representation in memory :

في لغة البرمجة تعرف المصفوفة كالاتي :

Int {*or any other type*} x [N]

Ex:

Int a[10]; float a[20];

وهذا يعني تعريف هيكل بياني يستوعب مجموعة من العناصر البيانية عددها N مثلا باسم بياني واحد مثال X ويستخدم الدليل [index] للوصول الى العنصر البياني المطلوب وتتراوح قيمة الدليل (1=<i<=n) وبموجب هذا التعريف يحدد المعالج مناطق ذاكرة متعاقبة لاستيعاب مجموعة العناصر البيانية للمصفوفة ويكون عنوان الذاكرة الاول مخصصا للعنصر الاول في المصفوفة وهو ما يطلق عليه عنوان البداية (Base address, BA). يستخدم معالج اللغة Compiler المعادلة التالية للوصول الى بيانات المصفوفة في الذاكرة :

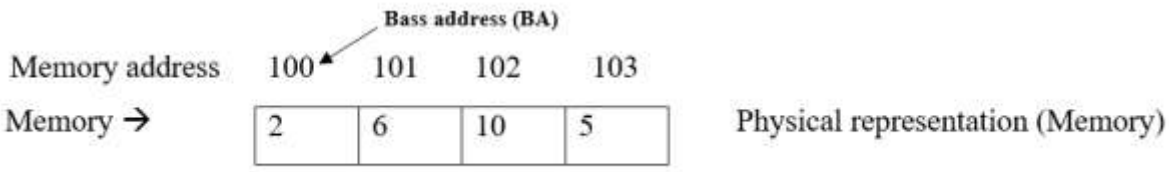
Location (x[i]) = Base Address + (i-1) (1)

حيث i يمثل موقع العنصر المطلوب

Example:

a [4] = {2,6,10,5};

a[1]=2, a[2]=6, a[3]=10, a[4]=5 Logical representation (programmer)



لذلك لو فرضنا اننا نريد طباعة العنصر الثالث من عناصر المصفوفة اعلاه فان المعالج سيستخدم المعادلة رقم (1) للوصول الى العنصر المطلوب في الذاكرة وكما يلي

Loc (a[i]) = BA+ (i-1) → Loc (a[3])=100+(3-1) → Loc(a[3]) = 302

Ex 2: Let int x [n] any array. Compute the memory address of the element x [4] when the bass Address is 500?

Solution:

i=4

Location(x [4]) =500+ (4-1)

=500+3=503

* فعندما يتضمن البرنامج اية اشارة او تعامل مع عناصر المصفوفة في اي ايعاز مثل cin>>x[i] او cout<< x[i] او غيرها فان المترجم يعتمد المعادلة 1 المشار اليها اعلاه لتحديد الموقع المطلوب .

تمثيل المصفوفة الثنائية في الذاكرة Representation of 2D array in memory :

في لغة البرمجة تعرف المصفوفة الثنائية كالاتي ،

int {or any other type} A [M][N]

Ex:

Int A [5][10];

Float A [4][6];

وهذا يعني تعريف هيكل بياني اسمه (A) يستوعب مجموعة من العناصر البيانية عددها (M*N) ويستخدم دليلين للوصول الى العناصر البياني المطلوب وهما

1=<i<=M لتحديد الصف الذي فيه العنصر

1=<j<=N لتحديد العمود الذي فيه العنصر

فمثلا العنصر A[3][5] حيث i=3 وj=5

وهذا يعني ان العنصر يقع في الصف الثالث والعمود الخامس .

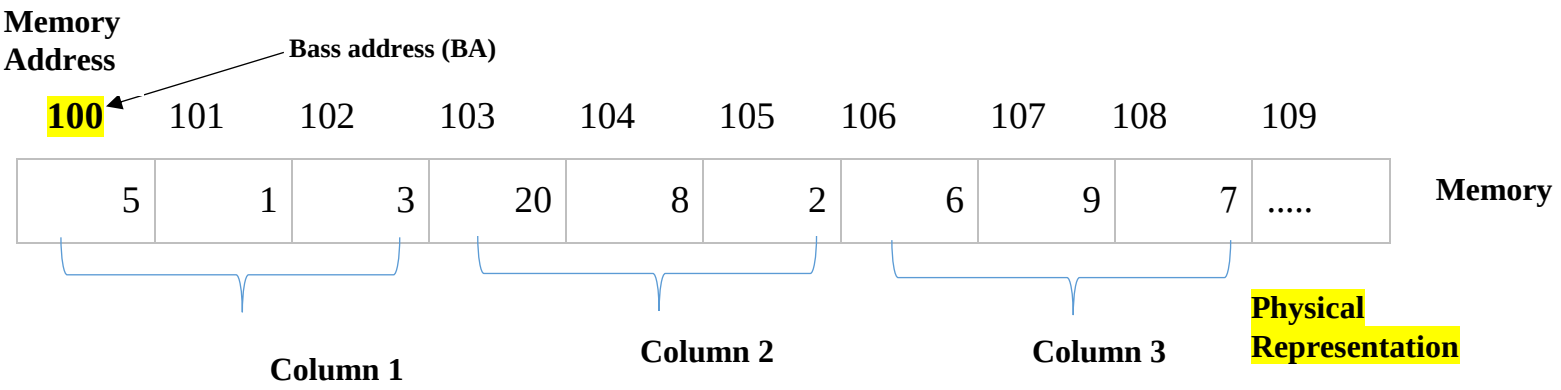
Ex: a[3][4] = $\begin{matrix} 5 & 20 & 6 & 5 \\ 1 & 8 & 9 & 1 \\ 3 & 2 & 7 & 4 \end{matrix}$ → = $\begin{matrix} a11 & a12 & a13 & a14 \\ a21 & a22 & a23 & a24 \\ a31 & a32 & a33 & a34 \end{matrix}$ → Logical representation

المثال اعلاه يعكس التمثيل المنطقي للمصفوفة الثنائية من وجهة نظر المبرمج. يعتمد مترجم اللغة (compiler) احدى الطريقتين الاتيتين لتمثيل المصفوفة الثنائية البعد فيزيائيا في الذاكرة وكما موضح ادناه .

1- طريقة الاعمدة Column method :

وهذه الطريقة مستخدمة في لغة الفورتران والبيسك.

توخذ عناصر العمود الاول (j=1) للمصفوفة وتخزن في مواقع متعاقبة في الذاكرة ثم يتم اخذ عناصر العمود الثاني ليتم خزنها بعد اخر عنصر من عناصر العمود الاول وهكذا وصولا الى العمود الاخير في المصفوفة كما موضح بالشكل ادناه.



وعليه فان احتساب موقع العنصر A[i][j] يكون وفق المعادلة التالية :

$$\text{Location}(A[i][j]) = \text{Base Address} + M \times (j-1) + (i-1) \qquad (2)$$

حيث ان M تمثل عدد الصفوف الكلي للمصفوفة

Ex: Let int s [3] [4] any Array, compute the location of the element [3] [2] when Base Address is (100).

Sol:

i=3, j=2, M=3, N=4

Location (s [3] [2]) =100+ 3 (2-1) + (3-1)

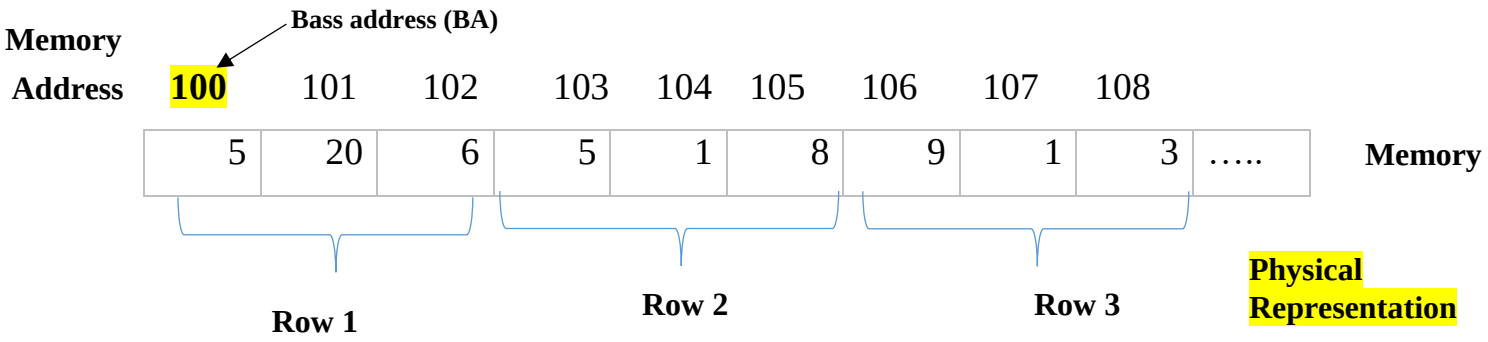
= 100+ 3+2 = **105**

2- طريقة الصفوف Row method:

وهذه الطريقة مستخدمة في لغة باسكال ،كوبول ، C++

حيث توخذ جميع عناصر الصف الاول للمصفوفة الثنائية وتخزن في الذاكرة ابتداء من موقع البداية (Base Address).

ثم توخذ جميع عناصر الصف الثاني للمصفوفة وتخزن في الذاكرة بعد اخر عنصر من عناصر الصف الاول وهكذا الى نهاية الصفوف.



ولهذا فان احتساب موقع العنصر A[i][j] يكون وفق المعادلة التالية :

Location (A[i] [j]) = Base Address + N * (i-1) + (j-1) (3)

حيث ان N تمثل عدد الاعمدة الكلي للمصفوفة

Ex: Let int A [3] [4]

Compute the location of the element A [2] [4] ?

When the base Address is 100.

Sol:

i=2, j=4 , M=3 , N =4

Location (A [2] [4]) = 100+ 4 * (2-1) + (4-1)

= 100 + 4 + 3 = **107**

Stack

Introduction

A stack is a linear data structure that follows a particular order in which operations are performed. The order is **Last In, First Out (LIFO),** meaning that the most recent element added to the stack will be the first one to be removed. This concept is frequently visualized as a stack of plates where you add plates to the top and remove them from the top.

ألمكدس عبارة عن بنية بيانات خطية تتبع ترتيبًا معينًا هو **"آخر من يدخل، يكون أول من يخرج"** (LIFO)، مما يعني أن أحدث عنصر يضاف إلى المكدس سيكون أول ما يتم إزالته. غالبًا ما يتم تصور هذا المفهوم على أنه مكدس من الألواح حيث تضيف ألواحًا إلى الأعلى وتزيلها من الأعلى أيضًا.

ملاحظة : يتم استخدام مؤشر نسميه (top) للدلالة على اخر عنصر تم ادخاله الى المكدس

Characteristics of Stack **خصائص المكدس**

1. **LIFO**: Last In, First Out.
2. **Basic Operations** **العمليات الاساسية للمكدس**:
   - **Push**: Adds an element to the top of the stack. **اضافة عنصر للمكدس**
   - **Pop**: Removes the top element of the stack. **سحب عنصر من المكدس**
   - **isEmpty**: Checks if the stack is empty. **تدقيق خلو المكدس من العناصر**
   - **isFull**: Checks if the stack has reached its maximum capacity. **تدقيق امتلاء المكدس بالعناصر**

Algorithms of Stack Operations

Push Operation

The push operation adds an element to the top of the stack. If the stack is full, it indicates a stack overflow condition. عملية اضافة عنصر للمكدس

Algorithm:

1. Check if the stack is full. If true, display 'Stack Overflow' and exit.
2. If not full, increment the top pointer to point to the next empty space.
3. Add the new element at the position indicated by the top pointer.

1. تحقق إذا كان المكدس ممتلئًا. إذا كان ممتلئًا، أعرض رسالة "المكدس ممتلئ" ولا تقم بإضافة العنصر.
2. إذا لم يكن المكدس ممتلئًا، زد مؤشر (top) بمقدار 1.
3. ضع العنصر الجديد في الموقع المحدد بواسطة مؤشر القمة.

Pop Operation

The pop operation removes the element from the top of the stack. If the stack is empty, it indicates a stack underflow condition. عملية سحب عنصر من المكدس

Algorithm:

1. Check if the stack is empty. If true, display ***'Stack Underflow'*** and exit.
2. If not empty, retrieve the element at the top pointer.
3. Decrease the top pointer to indicate the removal.

1. تحقق إذا كان المكدس فارغًا. إذا كان فارغًا، أعرض رسالة ***'Stack Underflow'*** ولا تقم بإزالة أي عنصر.
2. إذا لم يكن المكدس فارغًا، قم بإزالة العنصر الموجود في الموقع المحدد بواسطة مؤشر القمة.
3. أنقص مؤشر القمة بمقدار 1.

التحقق من خلو المكدس من العناصر Check Empty Stack (isEmpty)

The isEmpty operation checks if there are any elements in the stack.

Algorithm:

1. If the top pointer is -1, return true (stack is empty).
2. Otherwise, return false (stack is not empty).

1. تحقق إذا كان مؤشر القمة (top) أقل من 0.
2. إذا كان مؤشر القمة أقل من 0، فإن المكدس فارغ؛ أعرض "نعم."
3. إذا لم يكن مؤشر القمة أقل من 0، فإن المكدس غير فارغ؛ أعرض "لا."

التحقق من امتلاء المكدس بالعناصر Check Full Stack (isFull)

The isFull operation checks if the stack has reached its maximum capacity.

Algorithm:

1. If the top pointer is equal to MAX-1 (maximum capacity), return true (stack is full).
2. Otherwise, return false (stack is not full).

1. تحقق إذا كان مؤشر القمة أكبر من أو يساوي الحجم الأقصى للمكدس ناقص 1.
2. إذا كان مؤشر القمة يساوي أو أكبر من الحد الأقصى، فإن المكدس ممتلئ؛ أعرض "نعم."
3. إذا لم يكن مؤشر القمة يساوي أو أكبر من الحد الأقصى، فإن المكدس غير ممتلئ؛ أعرض "لا."

تطبيقات المكدس Stack applications

يُعد المكدس (Stack) أداة فعّالة في البرمجة والانظمة الحاسوبية ومنها معالجة التعابير الحسابية حيث يُستخدم في تحويل التعابير الحسابية من صيغة معينة إلى أخرى، وكذلك في تقييم هذه التعابير

صيغ تمثيل التعابير الحسابية :

- **Infix** : الصيغة التي نستخدمها عادةً في الكتابة التقليدية، حيث تكون العملية بين المعاملين، مثل A+B
- **Postfix** : الصيغة التي يكتب فيها المعاملات أولاً ثم العملية. مثل AB+
- **Prefix : الصيغة التي يكتب فيها المعاملات بعد العملية الحسابية مثل AB+**

1- استخدام المكدس في تحويل التعابير الحسابية من صيغة Infix إلى صيغة Postfix

لماذا نحتاج إلى التحويل من Infix إلىPostfix ؟

التحويل من Infix إلى Postfix له عدة فوائد، منها:

1. **بساطة التقييم** :الصيغة Postfix لا تتطلب أولويات العمليات أو استخدام الأقواس، مما يجعل تقييم التعابير الحسابية أكثر سهولة.
2. **تجنب الأقواس** :فيPostfix ، لا توجد حاجة إلى استخدام الأقواس للتعبير عن ترتيب العمليات.
3. **كفاءة المعالجة في الحواسيب** :تُستخدم Postfix بشكل واسع في الأنظمة المحوسبة بسبب قدرتها على تسريع عمليات الحساب باستخدام المكدس.

خوارزمية تحويل التعابير من Infix إلى Postfix باستخدام المكدس

لتحويل تعبير من Infix إلى Postfix باستخدام المكدس، نستخدم خوارزمية تعتمد على ترتيب العمليات.

1- ت**هيئة المكدس وسلسلة Postfix**

- ابدأ بسلسلة فارغة Postfix ستحتوي على النتيجة النهائية.

- استخدم مكدسًا لتخزين العمليات.

2- قراءة كل رمز في التعبيرInfix من اليسار الى اليمين

3- التعامل مع كل رمز حسب نوعه

- **إذا كان الرمز معاملًا:(Operand)**
  - ✓ أضفه مباشرة إلى سلسلة .Postfix
- **إذا كان الرمز عملية حسابية:(Operator)**
  - ✓ تحقق من ترتيب العمليات، وأضف العمليات الأعلى أولويةً في المكدس إلى سلسلة Postfixحتى يصبح المكدس فارغًا أو يحتوي على عملية ذات أولوية أقل.
  - ✓ بعد ذلك، ضع العملية الحالية في المكدس.
- **إذا كان الرمز قوسًا مفتوحًا)**
  - ✓ ضع القوس في المكدس.
- **إذا كان الرمز قوسًا مغلقًا (**

4- استمر في إزالة العمليات من المكدس وإضافتها إلى سلسلة Postfixحتى تصل إلى القوس المفتوح المقابل ويتم اخراجه من المكدس ايضا

5- **التعامل مع العناصر المتبقية في المكدس**

- بعد الانتهاء من قراءة كل الرموز، استمر في إخراج العمليات المتبقية في المكدس وأضفها إلى .Postfix

جدول الأسبقية للعمليات الرياضية

العملية	الاسبقية
الأس ,NOT	3
الضرب القسمة AND	2

OR الجمع الطرح	1

Ex: Convert the following infix expression into a postfix expression using stack

Infix: `A + B * (C - D)`

الخطوة	الرمز	العملية	حالة المكدس	سلسلةPostfix
1	A	معامل	-	A
2	+	عملية	+	A
3	B	معامل	+	A B
4	*	عملية	* +	A B
5	(	قوس مفتوح	(* +	A B
6	C	معامل	(* +	A B C
7	-	عملية	- (* +	A B C
8	D	معامل	- (* +	A B C D
9	)	قوس مغلق	* +	A B C D -
10	انتهاء	إخراج باقي العمليات	-	A B C D - * +

Ex: Convert the following infix expression to a postfix expression using a stack

Infix: `(A + B) * (C - D) / E`

الخطوة	الرمز	العملية	حالة المكدس	سلسلةPostfix
1	(	قوس مفتوح	(	-
2	A	معامل	(	A
3	+	عملية	+ (	A
4	B	معامل	+ (	A B
5	)	قوس مغلق	-	A B +
6	*	عملية	*	A B +
7	(	قوس مفتوح	* (	A B +
8	C	معامل	* (	A B + C

A B + C	* (-	عملية	-	**9**
A B + C D	* (-	معامل	D	**10**
A B + C D -	*	قوس مغلق	)	**11**
A B + C D -	* /	عملية	/	**12**
A B + C D - E	* /	معامل	E	**13**
A B + C D - * E /	-	إخراج باقي العمليات	انتهاء	**14**

يصبح التعبير بصيغة Postfix كالتالي **A B + C D – * E /**

2- حساب صيغة Postfix باستخدام المكدس

بمجرد تحويل التعبير إلىPostfix ، يمكن حساب ناتجه باستخدام المكدس:

الخوارزمية:

1. مر عبر كل رمز في تعبير.Postfix
2. إذا كان الرمز معاملًا، ضعه في المكدس.
3. إذا كان الرمز عملية، نفّذ العملية على آخر عنصرين في المكدس.
4. استمر حتى تصل إلى نهاية التعبير، وستكون النتيجة النهائية هي العنصر الوحيد المتبقي في المكدس.

Example: Compute the following postfix expression

احسب التعبير التالي **Postfix:** **6 2 3 + - 3 8 2 / + ***

الحل:

حالة المكدس	**الوصف**	**الرمز**	**الخطوة**
6	ضع 6 في المكدس	6	**1**
6, 2	ضع 2 في المكدس	2	**2**
6, 2, 3	ضع 3 في المكدس	3	**3**
6, 5	أجرِ عملية 2 + 3 = 5	+	**4**
1	أجرِ عملية 6 – 5 = 1	-	**5**
1, 3	ضع 3 في المكدس	3	**6**
1, 3, 8	ضع 8 في المكدس	8	**7**

1, 3, 8, 2	ضع 2 في المكدس	2	**8**
1, 3, 4	أجرِ عملية 8 / 2 = 4	/	**9**
1, 7	أجرِ عملية 3 + 4 = 7	+	**10**
7	أجرِ عملية 1 * 7 = 7	*	**11**

بعد تنفيذ كل العمليات، ستكون النتيجة النهائية هي العنصر الوحيد المتبقي في المكدس، وهو 7

H.W: convert the following infix to postfix then find the result of postfix

3 + 4 * 2 / (1 - 5) ^ 2 ^ 3

DATA STRUCTURES AND ALGORITHMS

التخصيص الخزني storage allocation:

1-التخصيص الخزني الثابت Static storage allocation

يعتبر ابسط طرق خزنالبيانات باستخدام القوائم الخطية حيث يتم خزن البيانات في مواقع خزنية متتابعة (متسلسلة) في ذاكرة الحاسوب . يمكن الوصول لاي عنصر اذا عرفنا موقع العنصر الاول الذي يمثل عنوان البداية) Base Address (حيث ان مواقع العناصر الاخرى يمكن حسابها من خلال موق البداية .مثلا العنصر (K) سيكون موقعه مباشرة بعد العنصر (K-1) وهكذا بقية لعناصر.

المزايا:

1-سهولة التمثيل والتطبيق
2-كفاءة عالية في الوصول العشوائي للبيانات
3-مناسب جدا عند التعامل مع المكدس والطابور

المساوئ:

1. صعوبة تنفيذ عمليات الاضافة والحذف
2. يتطلب التعريف المسبق وتحديد عدد العناصر المطلوب خزنها

2-التخصيص الخزني الديناميكي Dynamic storage allocation

في طريقة الخزن هذه يتم استخدام رابط **link** او مؤشر لكل عنصر يحتوي على عنوان موقع العنصر التالي لذلك لا توجد ضرورة لخزن البيانات في مواقع متعاقبة (متسلسلة) في الذاكرة بل يمكن خزن العنصر(البيانات) في اي موقع في الذاكرة .لذلك كل عنصر يحتوي على جزئيين هما:
الجزء الاول : حقل يحتوي على البيانات (Data)
الجزي الثاني : حقل يحتوي على عنوان موقع العنصر التالي (**link**)

المزايا:

- سهولة تنفيذ عمليات الاضافة والحذف لأي من خلال تحديث قيمة حقل المؤشر الذي يبين عنوان العنصر

التالي او السابق.

المساوئ:

- يحتاج الي مساحة خزنيه اكبر لتمثيل حقل المؤشر اضافة الى البيانات الأساسية.

المقارنة بين الخزن الثابت والخزن الديناميكي

1- **المساحة الخزنية** : Amount of storage ان اسلوب الخزن الديناميكي يحتاج الي مساحة خزنية اكبر لان كل عنصر في الهيكل البياني يحتاج الى مؤشر (اي موقع خزني اضافي) يحتوي على عنوان موقع العنصر التالي وهكذا جميع العناصر.

2- **عمليات الاضافة والحذف** : ان هذه العمليات اكثر سهولة في الخزن الديناميكي لانها لا تتطلب سوى تحديث قيمة المؤشر عند الاضافة او الحذف .اما في الخزن الثابت فالعملية اكثر تعقيدا لانها تتطلب عملية تزحيف(shifting) للعناصر.

3- **الوصول العشوائي للعناصر** : Random Access ان اسلوب الخزن الثابت يعتبر افضل في الوصول العشوائي لاي عنصر من عناصر الهيكل البياني مباشرة (k-th element from the start) اما في الخزن الديناميكي فانه يتطلب البدء من اول عنصر ثم العناصر التالية بالتتابع لحين الوصول للعنصر المطلوب.

4- **الدمج والفصل** : Merge and Split في الخزن الديناميكي تكون هذه العمليات اسهل تنفيذا وذلك بتغير قيمة المؤشر للعناصر(العقد) في مواقع الدمج او الفصل .اما في الخزن الثابت فالعمل اكثر تعقيدا اذ قد يتطلب تزحيف العناصر واعادة تنظيم الخزن Reorganization .

مفهوم المراجع References

- المراجع (**References**) هي العناوين التي يتم إعطائها لأي شيء (مثل المتغيرات, المصفوفات و الكائنات) يتم تعريفه في الذاكرة عند تشغيل البرنامج.

- عناوين الأشياء التي يتم تخصيص مساحة لها في الذاكرة, يتم وضعها بأسلوب **Hexadecimal** حيث تجد أغلب العنواين تحتوي على أرقام و أحرف كالتالي **0xd5ef87c**.

- الوصول للأشياء الموجودة في الذاكرة أمر مهم جداً حيث يجعلك قادر على تقليل المساحة التي يحتاجها برنامجك من الذاكرة. كما أنه قد يجعل حجم الكود أصغر حيث أنك تستطيع الوصول للأشياء الموجودة فيها بشكل مباشر و هذا الأمر ستلاحظه إن كنت تعمل في مشاريع ضخمة.

- إمكانية الوصول للأشياء الموجودة في الذاكرة هو أهم ما يميز لغة **C++** عن باقي اللغات التي لا يمكن فيها ذلك كلغة جافا و لغة بايثون.

- للوصول إلى عناوين الأشياء الموجودة في الذاكرة نستخدم العامل **&** الذي يقال له **Address Operator**.

طباعة عناوين الأشياء الموجودة في الذاكرة في C++

لطباعة عنوان المساحة المخصصة لأي متغير تم تعريفه في الذاكرة نضع & قبل إسمه كما سنرى في المثال التالي.

```
#include <iostream.h>
int main()
{
   // 5 هنا قمنا بتعريف متغير إسمه x و قيمته
   int x = 5;
      // هنا قمنا بطباعة عنوان المساحة التي تم تخصيصها للمتغير x في الذاكرة
   cout << "Address of x in memory: " << &x;
      return 0;
}
```

سنحصل على نتيجة تشبه النتيجة التالية عند التشغيل.

```
Address of x in memory: 0x61fe1c
```

مفهوم المؤشرات C++

تعرفنا ان المراجع (References) تستخدم بهدف جعلنا قادرين على الوصول إلى الأشياء المعرفة في الذاكرة بشكل مباشر.
المؤشرات (Pointers) تستخدم أيضاً للوصول لأي شيء يتم تعريفه في الذاكرة و هي تعطينا المزيد من المزايا في التحكم.

الفرق بين المرجع و المؤشر

الفرق الأساسي بينهما هو أن المؤشر يقوم بحجز مساحة في الذاكرة لتخزين عنوان الشيء الذي يؤشر إليه. بالإضافة إلى ذلك, فإن المؤشر يمكنه الإشارة لأي شيء موجود في الذاكرة في أي وقت و لست مجبراً على تحديد الشيء الذي يشير إليه لحظة إنشاءه.

تعريف مؤشر في C++

لتعريف مؤشر جديد نستخدم الرمز * مع الإشارة إلى أن نوع المؤشر يجب أن يكون نفس نوع الشيء الذي سيشير له في الذاكرة.

إذا اردنا تعريف مؤشر نوعه int و إسمه x فعندنا ثلاث خيارات كالتالي.

```
// الأسلوب الأول و الذي يعتبر الأكثر استخداماً
int* x;

// الأسلوب الثاني
int *x;

// الأسلوب الثالث
```

```
int * x;
```

لتعريف مؤشر و جعله يشير لقيمة شيئ موجود في الذاكرة, يجب أن نقوم بتمرير عنوان هذا الشيء كقيمة للمؤشر و عندها سيصبح المؤشر

قادر على الوصول لقيمته و عنوانه كما سنرى في المثال التالي.

```
#include <iostream>

using namespace std;

int main()
{
   // "Arabic" و قيمته language هنا قمنا بتعريف متغير إسمه
   string language = "Arabic";

   // في الذاكرة language هنا قمنا بتعريف مؤشر لعنوان المتغير
   string* ptr = &language;

   // language هنا قمنا بطباعة قيمة المتغير
   cout << "language = " << language << endl;

   // في الذاكرة language هنا قمنا بطباعة عنوان المتغير
   cout << "Address of language = " << &language << endl;

   // في الذاكرة ptr هنا قمنا بطباعة عنوان المؤشر
   cout << "Address of ptr in memory = " << &ptr << endl;

   // language هنا قمنا بطباعة القيمة الموجودة في المؤشر ptr و التي هي عنوان المتغير
   cout << "Value of ptr in memory = " << ptr << endl;

   // language هنا قمنا بطباعة القيمة التي يشير إليها عنوان المؤشر ptr في الذاكرة و التي هي قيمة المتغير
   cout << "Value that ptr point to = " << *ptr;

   return 0;
}
```

سنحصل على نتيجة تشبه النتيجة التالية عند التشغيل.

```
language = Arabic
Address of language = 0x61fde0
Address of ptr in memory = 0x61fdd8
Value of ptr in memory = 0x61fde0
Value that ptr point to = Arabic
```