الفصل السادس : المكدسات و الأرتال
Stacks and Queues

٦-١ مفهوم المكدس

تعتبر المتجهات (Arrays) و القوائم المترابطة (Linked Lists) ، و بنى المعطيات الأخرى مثل المكدس و الرتل بنى ملائمة لتنظيم البيانات التي نجدها في تطبيق قاعدة البيانات . و هي عادة ما تستخدم في السجلات الوظيفية و سجلات الجرد و البيانات المالية ، كما أن استخدام هذه البنى من شأنه أن يساهم في تيسير عملية الوصول إلى البيانات و التعامل معها سواء بالإدخال أو بالحذف أو البحث عن عناصر معينة .
و تأتي على-الجانب الآخر- بنى المعطيات و طرق التعامل التي سنتناولها في هذا الفصل و التي هي أكثر من مجرد أدوات في يد المبرمج . فهي في أساسها عوامل مساعدة أكثر منها وسائل لتنظيم البيانات . و لذلك فإن العمر الافتراضي لها قصير حيث إنها عادة ما تستخدم لأداء دور معين في تنفيذ عملية ما داخل البرنامج . و عند إنجاز هذه المهمة ، فإنها تختفي .

٦-١-١ أسلوب الوصول إلى البيانات في المكدسات و الأرتال

في تركيب المتجهة ، لا يوجد أية قيود على التعامل مع البيانات ، فقد كان من الممكن الوصول إلى أي عنصر من عناصر البيانات و بصورة مباشرة بشرط معرفة رقم الفهرس لهذا العنصر أو بالبحث في جميع خانات التركيب حتى يتم العثور عليه . أما في تراكيب المكدسات و الأرتال على الجانب الآخر ، فإن عملية الوصول هذه تكون محدودة بشرط معين و هو: أنه لا يمكن حذف أو عرض أكثر من عنصر واحد في المرة الواحدة .

٦-١-٢ كيفية تنظيم البيانات في تراكيب البيانات المكدسة

في تركيب المكدسات ، لا يمكن الوصول إلا لعنصر واحد فقط ، و هو آخر عنصر تم إدخاله إلى التركيب . و لو تم حذف هذا العنصر ، فإنه من الممكن الوصول إلى العنصر الذي يليه

مباشرة وهكذا . و يعد ذلك ميزة كبيرة في الكثير من البرامج . و سنرى في هذا الفصل كيف يمكن استخدام البيانات المكدسة في تحويل التعابير الحسابية (Arithmetic Expressions) من نمط النظامي إلى نمط الملحقة ونمط المصدرة و كذلك في تقييم (Evaluation) التعابير الحسابية الممثلة بنمط الملحقة.

و علاوة على ذلك ، يساعد تركيب البيانات المكدسة على تنفيذ العديد من العمليات البرمجية في بعض بنى المعطيات المعقدة مثل تنفيذ أساليب التنقل على الأشجار الثنائية (Binary Search) و تنفيذ البحث بالعمق أولا (Depth First Search) على البيان (Graph) .

و هناك العديد من معالجي البيانات (Microprocessors) التي تستخدم أسلوب التركيب المبني على فكرة البيانات المكدسة . فعند استدعاء دالة (Function) معينة ، يتم وضع العنوان و معامل الإدخال الخاص بها داخل المكدس . و عند انتهاء تنفيذ الدالة ، يختفيان مرة أخرى . و بناء على ذلك ، فقد تم بناء عمليات البيانات المكدسة داخل معالج البيانات .

٦–١–٣ مثال قياسي عن المكدس

الآن ، لكي نفهم الفكرة التي بني عليها بنى المعطيات، المكدس ، دعنا نعرض المثالين التاليين: في المثال الأول ، نستند إلى ظاهرة شائعة لدى بعض الناس ، و هي أنهم عندما يتلقون بريداً عادة ما يقومون بوضعه على منضدة أو في سلة لحين أن يتاح لهم الوقت لقراءته ، و لكن ذلك بعد أن يكون قد تجمع لديهم قدر هائل من الرسائل . و على ذلك ، فعندما يعودون لقراءة هذه الرسائل ، فإنهم يبدؤون بفتح آخر رسالة قد وصلتهم و يقرؤونها . و عند الانتهاء منها ، يستمرون بفتح الرسالة التي تليها و هكذا حتى يصلوا لآخر رسالة موجودة في السلة ، و التي هي أول رسالة من حيث الوصول .

في الواقع ، تنطوي هذه الطريقة على عيب واحد : أنه لو لم يتم قراءة الرسائل الموجودة في أعلى المجموعة ، فمعنى ذلك أن الرسائل في آخر المجموعة ستظل مهملة لمدة شهور. مما لاشك فيه أن هذا الأسلوب في قراءة الرسائل لا يسري على الجميع . فقد يقرأ البعض – مثلا – الرسائل الأولى التي تم وصولها أولا . أو قد يخلط بعض هذه الرسائل معا قبل

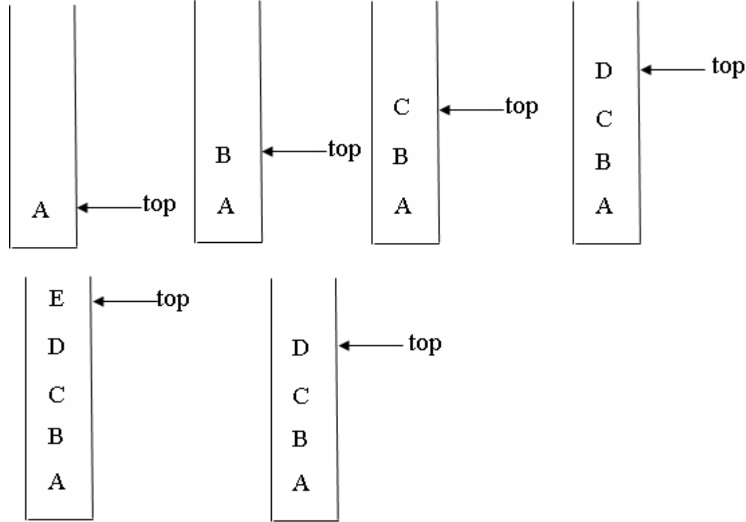
البدء بقراءتها ، و يبدأ في ترتيبها على حسب الأولوية . و اليوم ، لم يعد من الممكن أن نطلق على نظام البريد اسم المتراصات بما ينطوي عليه هذا المفهوم من معنى في علم الحاسوب . فلو بدأت قراءة الرسائل من أسفل المجموعة فذلك ما يطلق عليه اسم الرتل . و لو تم ترتيبها حسب الأولوية ، فذلك ما يطلق عليه اسم الأرتال ذات الأولوية (Priority Queue) . و هي جميعها بنى معطيات .

مثال قياسي آخر يوضح فكرة بنى المعطيات المكدسة ، و هو عبارة عن مجموعة من الأعمال التي قد تعرض على المرء للقيام بها في فترة ما . نفترض أنك تعمل في مشروع طويل الأجل و هو المشروع A . و في خلال عملك فيه جاءك مشروع آخر وهو مشروع B . و في أثناء عملك في المشروع B جاءك شخص لعقد اجتماع بخصوص موضوع ما C . و في خلال الاجتماع جاءت مكالمة هاتفية طارئة استغرقت بضع دقائق . بناء على ذلك ، فإن ما ستفعله هو أنك ستنتهي المكالمة الهاتفية أولا ، ثم تستأنف الاجتماع ، و بعدها تبدأ في إنهاء المشروع B ، ثم تعود مرة أخرى للمشروع A لاستكماله .

٦-٢ تعريف المكدس

هو قائمة مرتبة (خطية) من العناصر حيث يتم فيها الإدخال و الحذف من نهاية واحدة تدعى قمة المكدس top . فمثلا من أجل المكدس $S = (a_1, a_2, \dots, a_n)$ ، يكون a_1 العنصر الأسفل في المكدس و a_n العنصر الأعلى في المكدس ، ويكون العنصر a_i فوق العنصر a_{i-1} . $(1 < i < n)$

القيود على المكدس تستلزم إذا أدخلنا العناصر A, B, C, D, E إلى مكدس ، على الترتيب، عندئذ يكون E العنصر الأول الذي نستطيع حذفه من المكدس . الشكل (٦-1) يوضح متتالية المؤثرات الإضافة و الحذف . و بما أن كون آخر عنصر أضيف إلى المكدس يكون أول عنصر حذف منه . لذا ، فإن آلية تخزين العناصر في المكدس تكون بأسلوب Last-In-First –Out (LIFO) .



الشكل (٦-١) : الإضافة و الحذف للعناصر في المكدس

٦-٣ تمثيل المكدس Stack Representation

توجد طريقتان لتمثيل المكدس : ١- باستخدام المتجهات (one-dimensional array)
٢- باستخدام القوائم المترابطة (linked List)

٦-٣-١ تمثيل المكدس باستخدام المتجهات

تعتبر الطريقة الأسهل لتمثيل المكدس ، نقول ، stack[MAX_STACK_SIZE] حيث MAX_STACK_SIZE العدد الأعظمي للمدخلات . يخزن العنصر الأول أو الأسفل في stack[0] ، العنصر الثاني في stack[1] ، و العنصر ذو الدليل i يخزن في stack[i-1] كما نربط بالمتجهة متغير top الذي يتضمن دليل قمة المكدس . نجعل قيمة المتغير top مساوية لــ 1- للإشارة بأن المكدس فارغ ، و تصبح بنى المعطيات اللازمة لتعريف المكدس كما يلي:

```
#define MAX_STACK_SIZE  100 /* maximum stack size*/
  typedef struct {
          int key;
         /* other fields */
             } element ;
  element stack[MAX_STACK_SIZE ];
         int top = -1;
```

٦-٤ العمليات على المكدس

- اختبار كون المكدس ممتلئاً (isfull)
- اختبار كون المكدس فارغاً (isempty)
- إضافة عنصر إلى مكدس (add)
- حذف عنصر من مكدس (delete)

١- يتم اختبار كون المكدس ممتلئاً بالدالة الإجرائية التالية :

```
boolean isfull(int top)
     {
  if (top == MAX_STACK_SIZE-1) return true;
    else return false;
                 }
```

٢- يتم اختبار كون المكدس فارغاً بالدالة الإجرائية التالية :

```
boolean isempty(int top)
     {
 if (top== -1) return true;
  else return false;
     }
```

٣- إضافة عنصر إلى مكدس يتم بالدالة الإجرائية التالية :

```
void add(int *top, element item)
{
 if (isfull (*top) == true){
```

```
         printf("error : stack is full");
         return ;
             }
        stack[++*top]=item;
                        }
```

٤- حذف عنصر من مكدس يتم بالدالة الإجرائية التالية :

```
element delete(int *top)
    {
  if (isempty (*top) == true){
     printf("error : stack is empty");
     return ;
       }
     return stack[(*top)--];
         }
```

٦-٥ أمثلة لاستخدام المكدس في تقييم التعابير الحسابية

توجد ثلاثة أنماط لتمثيل التعابير الحسابية :

١. النمط النظامي (infix) ، هو نمط يستخدم الأقواس لكتابة التعابير و فيه تقع كل عملية (operator) بين عواملها (operands) أو متحولاتها .
٢. النمط الملحقة (postfix)، هو نمط لا يستخدم أقواساً (free- parenthesis) في كتابة التعابير و فيه تتبع كل عملية عواملها مباشرة .
٣. النمط المصّدرة (prefix)، هو نمط لا يستخدم أقواساً (free- parenthesis) في كتابة التعابير و فيه تسبق كل عملية عواملها مباشرة.

الجدول التالي يبين أمثلة لتعابير حسابية مكتوبة بالأنماط الثلاثة :

Infix	postfix	Prefix
2+3*4	234*+	+2*34
a*b+5	ab*5+	+*ab5
(1+2)*7	12+7*	*+127
a*b/c	ab*c/	/*abc
(a/(b-c+d))*(e-a)*c	abc-d+/ea-*c*	*/a-b+cd*-eac
a/b-c+d*e-a*c	ab/c-de*+ac*-	-+-/abc*de*ac

فيما يلي أمثلة لاستخدام المكدس في تقييم تعبير حسابي مكتوب بنمط الملحقة وكذلك تحويل التعابير الحسابية من نمط النظامي إلى نمط الملحقة .

٦-٥-١ تقييم تعبير حسابي مكتوب بنمط الملحقة

لتقييم تعبير حسابي ممثل بنمط الملحقة باستخدام المكدس ، نقوم بقراءة التعبير من اليسار إلى اليمين محرف – محرف ، فكلما وجد عامل (operand) يوضع في المكدس حتى يتم إيجاد مؤثر (operator) عندئذ نحذف عاملي المؤثر من أعلى المكدس ، و من ثم ننجز العملية المناسبة باستخدام هذا المؤثر، و من ثم نضع النتيجة في المكدس . نستمر بهذا النمط حتى نحصل في النهاية على قيمة واحدة في المكدس و تكون هي نتيجة التقييم ، و من ثم نحذفه من المكدس.

مثال: المطلوب تقييم التعبير +*62/3-42 باستخدام المكدس

الحل: الجدول التالي يوضح عملية التقييم للتعبير المعطى :

Token	Stack			Top
	[0]	[1]	[2]	
6	6			0
2	6	2		1
/	6/2			0
3	6/2	3		1
-	6/2-3			0
4	6/2-3	4		1
2	6/2-3	4	2	2
*	6/2-3	4*2		1
+	6/2-3+4*2			0

لكتابة الدالة الإجرائية المسؤولة عن تقييم التعبير الحسابي الممثل بنمط الملحقة، نفترض أن التعبير الحسابي يتضمن فقط المؤثرات الثنائية : % ,/ ,* ,- ,+ و العوامل عبارة عن أرقام صحيحة ، هذا الافتراض يسمح لنا بتمثيل التعبير الحسابي كمتجهة محارف . كما أن العوامل

المخزنة في المكدس من نوع int . كما أنه يصرح عن المكدس كمتجهة عامة (global array) .

قبل البدء بكتابة الدالة الإجرائية ، يتم التصريح عن التالي كمتجهات عامة :

```
#define MAX_STACK_SIZE 100    /* maximum stack size */
#define MAX_EXPR_SIZE 100     /* maximum size of expression */
typedef enum {lparen, rparen, plus, minus, times, divide, mod, eos,
operand} precedence;
int stack [MAX_STACK_SIZE] ;
char expr[MAX_EXPR_SIZE];
```

حيث يشير الرمز eos الموجود في القائمة إلى end-of-string .

```
int eval (void)
{
  /* Ealuate a postfix expression, expr, maintained as a
      global variable . '\0' is the end of the expression.
      The stack and top of the stack are global variables .
      get-token is used to return the token type and the
       character symbol . operands are assumed to be single
       character digits */

precedence token;
char symbol;
int top = -1;
int op1, op2;
int n = 0; /* counter for the expression string */
token = get_token(&symbol, &n);
while (token!=eos) {
        if(token==operand)
            add(&top, symbol-'0'); /*stack insert
         else {
             /* remove two operands , perform operation , and return result
                to the stack */
                op2=delete(&top);     /*stack delete */
                 op1=delete(&top);
                 switch (token){
                       case plus : add(&top, op1+op2);
                                         break;
```

```
                case minus : add(&top, op1-op2);
                        break;
                case times : add(&top, op1*op2);
                        break;
                case divide : add(&top, op1/op2);
                        break;
                case mod : add(&top, op1%op2);
                        break;
            }
          }
token = get_token (&symbol , &n);
  }
return delete (&top); /* return result */
      }
```

الدالة eval() تتضمن كوداً لتقييم تعبير حسابي ممثل بنمط الملحقة. و بما أن العوامل (operands) تكون محارف بشكل أولي ، لذا يجب تحويلها إلى أعداد صحيحة . العبارة symbol-'0' الموجودة في الدالة تنجز هذه المهمة ، حيث العبارة تقوم بطرح ما يقابل القيمة '0' في جدول ASCII و التي تساوي 48 ممّا يقابل symbol في جدول ASCII .

أما الدالة المساعدة (auxiliary) get_token() تملك الصيغة التالية:

```
precedence get_token(char *symbol, int *n)
    {
/* get the next token, symbol is the character representation,
   which is returned, the token is represented by its enumerated
   value , which is returned in the function name */
          *symbol = expr[(*n)++];
          switch (*symbol) {
           case '(' : return lparen;
           case ')' : return rparen;
           case '+' : return plus;
           case '-' : return minus;
           case '*' : return times;
           case '/' : return divide;
           case '%' : return mod;
           case ' ' : return eos;
```

```
default:  return operand; /*no error checking,
           default is operand */
        }
     }
```

٦-٥-٢ التحويل من نمط النظامي إلى نمط الملحقة

لنصف خوارزمية تقوم بتحويل تعبير حسابي ممثل بنمط النظامي إلى نمط الملحقة معطاة بالخطوات التالية :

١. وضع كافة الأقواس في التعبير الحسابي .
٢. تحريك كل المؤثرات الحسابية بحيث تبدّل بأقواسها اليمينية .
٣. حذف كل الأقواس .

مثال

ليكن التعبير a/b-c+d*e-a*c عندما توضع كافة الأقواس فيه يصبح بالشكل :
((((a/b)-c)+(d*e))-(a*c)) و بتطبيق الخطوتين (٢) و (٣) نحصل على ab/c-de*+ac*-

ملاحظة

على الرغم من سهولة عمل الخوارزمية السابقة عندما يتم تنفيذها باليد ، تعتبر غير فعالة باستخدام الحاسب لأنها تتطلب مرحلتين: ١- قراءة التعبير وأقواسه ، ٢- تحريك المؤثرات إلى أقواسها اليمينية . و بما أن ترتيب عوامل تعبير حسابي يبقى نفسه في كل أنماط تمثيله ، لذلك ننجز عملية التحويل بمرحلة واحدة : بقراءة التعبير الممثل بنمط النظامي من اليسار إلى اليمين ، و أثناء عملية المسح ، تمرر العوامل إلى تعبير الخرج و يتم إخراج المؤثرات الممسوحة اعتمادا على أولويتها : أي أن المؤثرات ذات الأولوية الأعلى تخرج أولا ، و يتم حفظ المؤثرات حتى نعرف موقعها الصحيح في تعبير الخرج ليتم نقلها إليه لاحقا . يعتبر المكدس أفضل بنى المعطيات لتخزين هذه المؤثرات .

مثال (تعبير بسيط):

ليكن التعبير الحسابي البسيط a+b*c ، و كما نعلم أن تمثيله بنمط الملحقة : هو abc*+ .
الجدول التالي يوضح عملية التحويل هذه ، حيث العوامل تخرج مباشرة و المؤثرات + و * تحتاج لان تحفظ في المكدس. لأن المؤثرات ذات الأولوية الأعلى تخرج قبل المؤثرات الأقل

منها بالأولوية ، و ثم نحفظ في المكدس المؤثرات القادمة من تعبير الدخل مادام أولويتها أكبر من أولوية المؤثر في أعلى المكدس .

في مثالنا هذا يتم إخراج المؤثرات من المكدس عندما نصل إلى نهاية التعبير (eos) حيث يكون مؤثر الضرب في أعلى المكدس و هو الأعلى في الأولوية من مؤثر الجمع .

Token	Stack			Top	Output
	[0]	[1]	[2]		
a				-1	a
+	+			0	a
b	+			0	ab
*	+	*		1	ab
c	+	*		1	abc
eos				-1	abc*+

مثال (تعبير بأقواس):

ليكن التعبير الحسابي البسيط a*(b+c)*d ، و كما نعلم أن تمثيله بنمط الملحقة: abc+*d*.

الجدول التالي يوضح عملية التحويل هذه ، في هذه العملية يتم قراءة التعبير الحسابي من اليسار إلى اليمين ، العوامل يتم إخراجها إلى تعبير الخرج و المؤثرات الحسابية و الأقواس اليسارية يتم تكديسها حتى يتم إيجاد القوس اليميني في التعبير. عندئذ يتم إخراج المؤثرات من المكدس إلى تعبير الخرج حتى نصل أول قوس يساري . عندئذ نحذف القوس اليساري من المكدس (القوس اليميني لا يوضع في المكدس) . و بالتالي يبقى فقط المقدار d* في التعبير . و بما أن عمليتي الضرب لهما الأولوية نفسها ، عندئذ تخرج واحدة قبل d و الثانية تكدس و بعد ذلك تخرج من المكدس إلى تعبير الخرج بعد d.

Token	Stack			Top	Output
	[0]	[1]	[2]		
a				-1	a
*	*			0	a
(	*	(		1	a
b	*	(		1	ab
+	*	(	+	2	ab
C	*	(	+	2	abc
)	*			0	abc+
*	*			0	abc+*
d	*			0	abc+*d
eos	*			0	abc+*d*

قبل كتابة الدالة الإجرائية التي تنفذ عملية التحويل هذه ، نعرف متجهتين عامتين :الأولى تعرف أولوية المؤثرات المكدسة in-stack-precedence (isp) ، و الثانية تعرف أولوية المؤثرات الممسوحة في التعبير incoming precedence (icp) و كذلك المكدس بالشكل:

```
Prcecdence stack [MAX_STACK_SIZE];
int isp []={0, 19, 12, 12, 13, 13, 13, 0};
int icp[] ={20, 19, 12, 12, 13, 13, 13, 0};
```

بما أن قيمة متغير في قائمة enumerated يأخذ قيمة صحيحة توافق قيمة موقعه في القائمة ، فمثلا plus تأخذ قيمة 2 وبالتالي isp[plus]=isp[2] = 12 و بالتالي

```
   void postfix(void)
         {
           precedence token;
           char symbol ;
           int n=0;
           int top=0;
       stack[0]=eos;
for (token=get_token(&symbol, &n); token !=eos; token=
get_token(&symbol, &n)){
   if (token==operand)
   printf("%c",symbol);
   else if (token==rparen) {
```

```
  while(stack[top]!=lparen)
  print_token(delete(&top));
   delete(&top);
            }
     else {
      while(isp[stack[top]]>=icp[token])
         print_token(delete(&top));
         add(&top, token);
               }
         }
while (token = delete(&top))!=eos)
print_token(token);
}
```

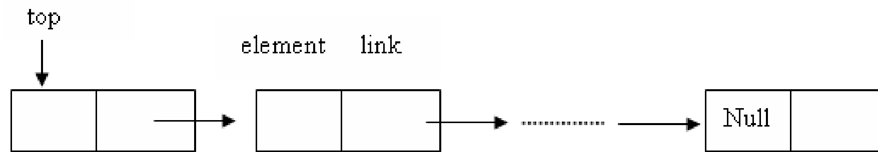
تحليل الدالة postfix()

ليكن n عدد tokens في التعبير الرياضي exp . $\Theta(n)$ يكن الزمن المستهلك لاستخلاص tokens و إخراجها. وكذلك هذا الزمن يستهلك في حلقتي while . الزمن الكلي المستهلك هنا هو $\Theta(n)$ مثل عدد tokens التي توضع في stack و تخرج من stack يكون خطياً في n لذلك زمن التنفيذ للخوارزمية هو $\Theta(n)$

٦-٣-٢ تمثيل المكدس باستخدام القوائم المترابطة

في هذه الطريقة يمثل المكدس بمجموعة من العقد (nodes) ، كل عقدة تتضمن حقلين : حقلاً للبيانات (data) : يتضمن حقل البيانات في كل عقدة عنصراً من المكدس و حقلاً للربط (link) يشير إلى العقدة التي يتضمن حقل البيانات فيها العنصر التالي من المكدس . كما أن حقل الربط في آخر عقدة يساوي الصفر (null) .

المتغير top يشير دوماً إلى أول عقدة في القائمة ، و تكون قيمته صفراً عندما يكون المكدس فارغ . الشكل (٦-٢) يبين بنية مكدس ممثل بقوائم المترابط .



الشكل (٦-٢): مكدس مترابط

يتم الإعلان عن مكدس بالشكل التالي:

```
typedef struct {
                    int key;
                    /* other fields */
                      } element ;
  typedef struct stack *stack_pointer;
  typedef struct stack{
            element data;
            stack_pointer link;
             };
           stack_pointer top;
```

نستطيع في هذا التمثيل كتابة العمليات التالية على المكدسات :

1- يتم اختبار المكدس الفارغ بالدالة الإجرائية التالية :

```
boolean isempty(stack_pointer top)
      {
    if (top== null) return true;
    else return false;
       }
```

٢- يتم اختبار المكدس الممتلئ بالدالة الإجرائية التالية :

```
boolean isfull(stack_pointer top)
      {
 stack_pointer temp =(stack_pointer) malloc (sizeof (stack));
    if (temp== null) return true;
    else return false;
       }
```

٣- إضافة عنصر إلى مكدس يتم بالدالة الإجرائية التالية :

```
void add(stack_pointer *top, element item)
        {
      /* add an element to the top of the stack  */
 stack_pointer temp = (stack_pointer) malloc (sizeof(stack))
  if (temp==null){
      printf("the memory is full");
      exit(1);
             }
    temp-> data=item;
     temp -> link = *top;
      *top=temp;
        }
```

٤- حذف عنصر من مكدس يتم بالدالة الإجرائية التالية :

```
element delete(stack_pointer *top)
        {
      stack_pointer temp = *top;
      element item;
       if (temp==null){
        printf("the stack is empty");
        exit(1);
             }
        item=temp->data;
        *top=temp->link;
        free(temp);
        return item;
           }
```

ملاحظة :

باستخدام طريقة القوائم المترابطة في تمثيل المكدس ، يمكننا تمثيل عدد من المكدسات بالشكل التالي :

```
#define MAX_STACKS 10 /* maximum number of stacks */
typedef struct {
                int key;
                /* other fields */
                  } element ;
```

```
typedef struct stack *stack_pointer;
typedef struct stack{
          element data;
          stack_pointer link;
           };
         stack_pointer top [MAX_STACKS];
```

نفرض أن شرط البدء لكل مكدس :

top[i] = null , 0 = < i< MAX_STACKS

و الشروط الحدية :

top[i] = null iff the ith stack is empty
isfull (temp)=true iff the memory is full

٦-٦ مفهوم الرتل Queue

يعتبر الرتل بنية من بنى المعطيات الأساسية شديد الشبه ببنية المكدس حيث لا يختلف عنه إلا في أن العنصر الأول في الإدخال هو نفسه العنصر الأول في الحذف . و ذلك ما يعرف First-In-First-Out (FIFO) .

تنطوي فكرة تنظيم البيانات في الرتل على فكرة صفوف المشاهدين التي تحتشد أمام شباك الحجز في أحد دور السينما . فأول من يحصل على تذكرة و يغادر الصف بكون الشخص الأول فيه ، بينما يكون الشخص الأخير هو آخر من يحصل على التذكرة و يغادر الصف . وبالنسبة لاستخدامات الرتل ، فهي لا تقتصر على كونها أداة برمجية فحسب ، بل تستخدم – أيضاً – لتمثيل مواقف واقعية مثل : صف العملاء الموجود داخل احد البنوك ، أو صف الطائرات الجاهزة للإقلاع ، و كذلك مجموعة البيانات المنتظر إرسالها عبر الانترنيت .

و فضلا عن ذلك يوجد العديد من الأرتال تؤدي مهام معينة في نظم التشغيل و الشبكات . فهناك – مثلا – الطابعة التي تنظم فيها الأعمال المنتظر طباعتها على الجهاز . و يترتب على ذلك أنك لا يمكنك أداء مهمتين في آن واحد على الجهاز: إذا كان هناك – مثلا – عمل ما يؤديه الحاسب ثم ضغطت على مفتاح معين لاستخدام معالج ، فلن تسجل هذه العملية على الجهاز و السبب في ذلك أن هذا الاستدعاء الأخير لمعالج النصوص سينتظر في رتل إلى

حين قراءته و التعامل معه . و بذلك تضمن بنية الأرتال الحفاظ على نسق و ترتيب تنفيذ العمليات المختلفة في الجهاز .

٦-٧ تمثيل الرتل Queue Representation

توجد طريقتان لتمثيل المكدس : ١- باستخدام المتجهات (one-dimensional array)
٢- باستخدام القوائم المترابطة (linked List)

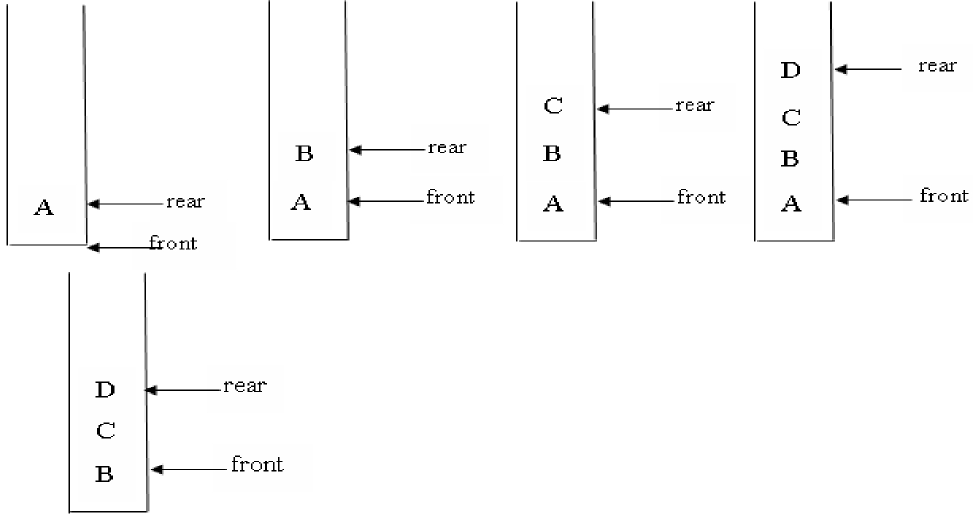
٦-٧-١ تمثيل الرتل باستخدام المتجهات

يعرف الرتل بأنه قائمة خطية (مرتبة) من العناصر حيث تتم الإضافة من جهة و الحذف من الجهة المعاكسة . بفرض لدينا الرتل التالي : $Q = (a_0, a_1, a_2, \ldots.., a_{n-1})$ ، حيث يكون العنصر a_0 في مقدمة الرتل (front element) ويكون العنصر a_{n-1} في مؤخرة الرتل ، و يكون العنصر a_{i+1} خلف العنصر a_i حيث $0 \le i < n$. إذا أدخلنا العناصر A, B, C, D, E إلى رتل ، على الترتيب ، عندئذ يكون A العنصر الأول الذي نستطيع حذفه من الرتل الشكل(٦-٣) يوضح متتالية تطبيق مؤشرات الإضافة و الحذف في الرتل.

تعتبر المتجهة الطريقة الأسهل لتمثيل الرتل ، نقول ، queue[MAX_QUEUE_SIZE] حيث MAX_ QUEUE _SIZE العدد الأعظمي للمدخلات . يخزن العنصر الأول في queue[0] ، العنصر الثاني في queue[1] ، و العنصر ذا الدليل i يخزن في queue[i-1] . كما نربط بالمتجهة متغيرين : الأول rear الذي يتضمن دليل آخر عنصر مضاف ، و الثاني front يتضمن دليل آخر عنصر تم حذفه من الرتل . نجعل قيمة كلا المتغيرين مساوية لــ 1- للإشارة بأن الرتل فارغ ، و تصبح بنى المعطيات اللازمة لتعريف الرتل كما يلي:

```
  #define MAX_QUEUE_SIZE  100 /* maximum queue size*/
typedef struct {
                 int key;
                 /* other fields */
                   } element ;
  element queue [MAX_ QUEUE _SIZE ];
```

```
int rear = -1;
int front = -1
```



الشكل(٦-٣) : الإضافة و الحذف للعناصر في الرتل

٦-٨ العمليات على الرتل

- اختبار كون الرتل فارغاً (isemptyq)
- اختبار كون الرتل ممتلئاً (isfullq)
- إضافة عنصر إلى رتل (addq)
- حذف عنصر من رتل (deleteq)

١- يتم اختبار الرتل الفارغ بالدالة الإجرائية التالية :

```
boolean isemptyq(int front, int rear)
    {
    if (front== rear) return true;
    else return false;
```

```
}
```

2- يتم اختبار الرتل الممتلئ بالدالة الإجرائية التالية :

```
boolean isfullq(int rear)
  {
  if (rear==MAX_QUUEUE_SIZE-1) return true;
  else return false;
    }
```

٣- إضافة عنصر إلى رتل يتم بالدالة الإجرائية التالية :

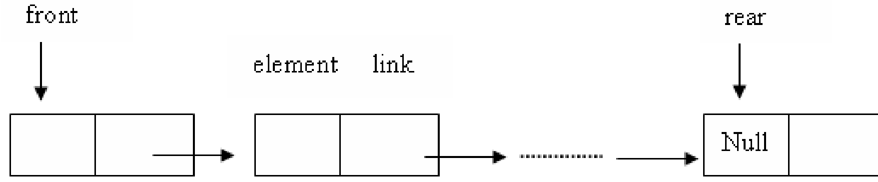
```
void addq(int *rear, element item)
   {
  if (isfullq (*rear) == true){
    printf("error : queue is full");
    return ;
    }
   queue[++*rear]=item;
     }
```

٤- حذف عنصر من رتل يتم بالدالة الإجرائية التالية :

```
element deleteq(int *front, int rear)
   {
  if (*front == rear){
    printf("error : queue is empty");
    return ;
    }
  return queue[++(*front)];
    }
```

٦-٧-٢ تمثيل الرتل باستخدام القوائم المترابطة

كما ذكرنا في تمثيل المكدس باستخدام القوائم المترابطة ، يمثل الرتل بمجموعة من العقد ، حيث المتغير front يشير دوما إلى أول عقدة في القائمة ، و المتغير rear يشير إلى آخر عقدة في القائمة و تكون قيمته صفراً عندما يكون الرتل فارغاً . الشكل (٦-٤) يوضح بنية رتل ممثل بقوائم الترابط .



الشكل (٦-٤) : رتل مترابط

يتم الإعلان عن رتل بالشكل التالي:

```
typedef struct queue * queue _pointer;
typedef struct queue {
            element data;
            queue _pointer link;
             };
           queue _pointer front, rear;
```

نستطيع في هذا التمثيل كتابة العمليات التالية على الرتل :

1- يتم اختبار الرتل فارغ بالدالة الإجرائية التالية :

```
 boolean isemptyq(queue _pointer front)
                      {
                    if (front== null) return true;
                    else return false;
                       }
```

2- يتم اختبار الرتل ممتلئ بالدالة الإجرائية التالية :

```
               boolean isfullq(queue_pointer rear)
                      {
     queue _pointer temp =( queue _pointer) malloc (sizeof (stack));
                    if (temp== null) return true;
                    else return false;
                       }
```

٣- إضافة عنصر إلى رتل يتم بالدالة الإجرائية التالية :

```
void addq(queue_pointer *front, queue_pointer *rear, element item)
        {
        /* add an element to the rear of the queue  */
  queue_pointer temp = (queue_pointer) malloc (sizeof(stack))
     if (temp==null){
        printf("the memory is full");
        exit(1);
           }
      temp-> data=item;
      temp -> link = null;
   if(*front !=null) *rear->link=temp;
     else *front = temp;
       *rear=temp;
        }
```

٤- حذف عنصر من رتل يتم بالدالة الإجرائية التالية :

```
element deleteq(queue_pointer *front)
     {
 /* delete an element from the queue
   queue_pointer temp = *front;
   element item;
    if (*front==null){
    printf("the queue is empty");
    exit(1);
        }
    item=temp->data;
    *front=temp->link;
    free(temp);
    return item;
      }
```

ملاحظة :

باستخدام طريقة القوائم المترابطة في تمثيل الرتل ، يمكننا تمثيل عدد من الأرتال بالشكل :

```
#define MAX_QUEUE 10 /* maximum number of queues */
 typedef struct queue * queue _pointer;
  typedef struct queue {
        element data;
```

```
        queue _pointer link;
         };
    queue _pointer front [MAX_QUEUE], rear [MAX_QUEUE];
```

نفرض أن شرط البدء لكل رتل :

front[i] = null ,   0 =< i< MAX_QUEUE

و الشروط الحدية :

front[i] = null   iff the ith queue is empty
isfull (temp)=true   iff the memory is full

٩-٩ الأرتال ذات الأولويات Priority Queues

تعتبر بنية الرتل ذات الأولوية من بنى المعطيات يمكن استخدامها كأداة هامة لتخزين عدد من المواقف المختلفة . فمثل الرتل (queue) السابق ، تحتوي الأرتال ذات الأولوية على بداية و نهاية (مقدمة و مؤخرة) ؛ و يكون إدخال العنصر من بداية التركيب بينما الحذف من نهايته . و لكن يضاف لذلك شيء واحد ؛ هو أن العناصر الموجودة تكون مرتبة تبعا" لمدخل القيمة لكل منها . لذا ، فإن العنصر ذا مدخل القيمة الأقل (أو في بعض التطبيقات ذا مدخل القيمة الأعلى) يكون دائما في المقدمة. و بذلك توضع العناصر التي يتم إدخالها في المكان المناسب داخل التركيب وفقا" لترتيب معين .
لا تختلف بنية الأرتال ذات الأولويات عن المكدسات و الأرتال السابق العرض لها ، فجميعها أدوات في يد المبرمج يستخدمها لشتى الأغراض في نظم التشغيل . ففي نظام التشغيل المتعدد المهام – مثلا- من الممكن تنظيم البرامج في شكل رتل له أولويات بحيث يكون البرنامج ذا الأولوية الأكبر هو البرنامج التالي في التنفيذ .
و فضلا" عن ذلك ، قد نرغب أحيانا في أن نصل إلى العنصر ذي القيمة الأقل (والذي قد يمثل الطريق الأقصر أو الأقل كلفة لعمل شيء معين) . و من ثم ، فإن هذا العنصر يكون صاحب الأولوية الأكبر في الوصول . و سيكون تنفيذ الأرتال ذات الأولويات على هذا

الأساس ؛ و إن كان هناك بعض المواقف الأخرى التي يحظى فيها العنصر صاحب مدخل القيمة الأكبر بالأولوية الأعلى .

بناء على ذلك ، ستوفر الأرتال ذات الأولوية في هذه الحالة الوصول السريع للعنصر المطلوب . و لكن الأمر لا يقف عند هذا الحد ، فقد نحتاج لهذا التركيب أيضا" لإدخال عنصر ما بسرعة كبيرة بعض الشيء . لذلك غالباً ما يتم تمثيل الرتل ذي الأولوية باستخدام متجهة.

ملاحظة: تنقسم الأرتال ذات الأولويات إلى نوعين :

تصاعدية الترتيب (ascending-priority queue) و

تنازلية الترتيب (descending –priority-queue) .

١. في الحالة الأولى يكون العنصر صاحب المدخل الأقل هو العنصر صاحب الأولوية الأكبر في الوصول . و من ثم ، يكون العنصر الأول في الحذف .
٢. في الحالة الثانية ، فإن العنصر صاحب الأولوية الأكبر يحمل مدخل القيمة الأعلى .

دائماً ما توجد العناصر ذات المداخل الأقل في آخر المتجهة (عند أعلى دليل) بينما توجد العناصر ذات المداخل الأكبر في أول المتجهة (عند الدليل 0) .

عند تطبيق الأرتال ذات الأولويات باستخدام مفهوم المتجهات لا يتم الحفاظ على ترتيب معين لعناصر البيانات . فالعناصر الجديدة يتم إدخالها بكل بساطة إلى آخر المتجهة مما يزيد من سرعة عملية الإدخال ، و لكن يبطئ في الوقت نفسه سرعة عملية الحذف لأنه لا بد من البحث عن العنصر صاحب القيمة الأصغر أولاً ؛ و هو أمر يتطلب فحص جميع العناصر و نقل نصفها تقريباً لملء الفجوة الناتجة عن عملية الحذف . و على أية حال ، فإنه يفضل إتباع الأسلوب السريع في الحذف .

يوضح البرنامج priorityQ .cpp كيفية مفهوم الأرتال ذات الأولويات على متجهات بسيطة مرتبة العناصر باستخدام لغة C++ .

```
#include <iostream>
#include <vector>
using namespace std;
////////////////////////////////////////////////////////////////
```

```
class priorityQ
{
//vector in sorted order, from max at 0 to min size-1
private :
int maxSize;
vector <double>queVect;
int nItems;
public:
//----------------------------------------------
priorityQ(int s) :  maxSize(s); nItems(0)   //constructor
          {queVect.resize(maxSize);}
//-----------------------------------------------
void insert (double item )    // insert item
 {
      int j;
if(nItems == 0)                            // if no items
queVect[nItems++] = item ;           // insert at 0
  else  {
     for(j = nItems-1; j >=0; j--) {   //start at end
        if(item > queVect[j])          // if new item larger
          queVect[j+1] = queVect[j];    // shift upward
  else      break;                          // if smaller, done shifting
     }  //    end for
 queVect[j+1] = item;                  // insert it
nItems++;
    }                                        // end else
   }                                         // end insert
//-----------------------------------------------------------------
double remove ()                   // remove minimum item
{ return queVect[--nItems];}
//-----------------------------------------------------------------
double peekMin()                 // peek at minimum item
{ return queVect[nItems-1];}
//-----------------------------------------------------------------
bool isEmpty()                 // true if queue is empty
{ return (nItems == 0;}
//-----------------------------------------------------------------
bool isFull()                  // true if queue is full
{ return (nItems == maxSize;}
//-----------------------------------------------------------------
     };       // end class PriorityQ
```

```
////////////////////////////////////////////////////////////////////////////////
int main ()
  {
   PriorityQ the PQ(5);                    //priority queue , size 5
        the PQ.insert(30);
        the PQ.insert(50);
        the PQ.insert(10);
        the PQ.insert(40);
        the PQ.insert(20);

while (!thePQ.isempty())
  {                            //sorted removals
double item = thePQ.remove();
cout<<item<<" ";  // 10, 20, 30, 40, 50
}          // end while
 cout<<endl;
return 0 ;
}             //end main()
```

حيث تم إدخال خمسة عناصر في main() و بترتيب عشوائي ، ثم حذفت جميع العناصر بعد ذلك و عرضت على الشاشة . و كان العنصر صاحب القيمة الأصغر هو الأول في عملية الحذف ، لذلك جاءت المخرجات على هذا النحو : 10, 20, 30, 40, 50

٦-١٠ تمارين الفصل السادس

تمرين١ : أعد كتابة الدالة ()postfix بحيث تعمل - إضافة للمؤثرات الحسابية و الأقواس- على المؤثرات التالية : == &&, !!, <<, >>, !=, <, >, <=, >= .

تمرين٢ : يعرف الرتل ذو النهايات المضاعفة (double-ended queue) و اختصاراً deque بأنه قائمة خطية يتم فيها إدخال العناصر و حذفها من مقدمة الرتل و كذلك من نهايته. والمطلوب :

١. قدم بنى المعطيات اللازمة لتمثيل هذا الرتل.
٢. اكتب دالة إجرائية لحذف عنصر من مقدمة الرتل.
٣. اكتب دالة إجرائية لحذف عنصر من نهاية الرتل.
٤. اكتب دالة إجرائية لإضافة عنصر في مقدمة الرتل.
٥. اكتب دالة إجرائية لإضافة عنصر في نهاية الرتل.
٦. اكتب دالة إجرائية لعكس ترتيب العناصر في هذا الرتل أي أن العنصر الأول يصبح الأخير و الثاني يصبح قبل الأخير .

تمرين٣ : اكتب خوارزمية تقوم بفحص عبارة نصية (أو عبارة رياضية) إذا كان فتح و إغلاق رموز الأقواس{ } , [ ] , ( ) الموجودة فيها مكتوباً بتسلسل صحيح أم لا، وذلك باستخدام مفهوم المكدسات.

مثلاً: العبارة التالية صحيحة:

{Math {example}: (2+3*[5-(8/4-2)]-(4/2+1))=x}

عبارة خاطئة لعدة أسباب:

{Math {example): (2+3*[5-(8/4-2)]-(4/2+1)=x]

عبارة خاطئة لعدة أسباب:

{Math {example}: (2+3*[5-(8/4-2]-(4/2+1))=x

تمرين٤ : باستخدام مفهوم المكدسات :

١. اكتب دالة إجرائية لتحويل تعبير حسابي ممثل بنمط المصدرة إلى نمط الملحقة ، و احسب زمن تنفيذها .
٢. اكتب دالة إجرائية لتحويل تعبير حسابي ممثل بنمط الملحقة إلى نمط المصدرة ، و احسب زمن تنفيذها .
٣. اكتب دالة إجرائية لتحويل تعبير حسابي ممثل بنمط الملحقة إلى نمط النظامي ممتلئ الأقواس (مثال: التعبير الحسابي ((a+b)+c) ممثل بنمط النظامي و ممتلئ الأقواس) واحسب زمن تنفيذها .
٤. اكتب دالة تقوم بتقييم تعبير حسابي ممثل بنمط المصدرة.

تمرين٥ : قدم خوارزمية غير عودية لمسألة أبراج هانوي معتمداً على مفهوم المكدسات. ثم احسب التعقيد الزمني للخوارزمية .