جَامعة
المَنارة
MANARA UNIVERSITY

كلية الهندسة قسم المعلوماتية

بنى معطيات 1

Data Structure 1

ا. د. علي عمران سليمان

محاضرات الأسبوع التاسع

الأشجار2

Tree 2

الفصل الثاني 2023-2024

محتوى الفصل

الأشجار1 + 2	Trees 1 +2
1- مقدمة	
2- الأشجار العامة.	
3- خوارزميات التجول عبر الشجرة.	
4- الأشجار الثنائية.	
5- تعريف شجرة البحث الثنائية كقاموس بيانات.	
6- تحقيق خوارزميات البحث.	
7- حساب أداء خوارزميات البحث.	
8 - تحقيق طرائق التبديل على شجرة بحث ثنائية.	
9 - تحقيق شجرة البحث الثنائية بلغة C++	

References

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

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

شجرة البحث الثنائي BST–بتعريف بسيط- عبارة عن بنية معطيات من نوع شجرة tree ثنائية, تملك الخصائص التالية،

- كل عقدة v من الشجرة مخصصة لتخزين مفتاح مميز (k,x) بحيث إن:
- المفاتيح المخزنة في العقد التي في الشجرة الفرعية اليسارية لـ v أصغر من k.
- المفاتيح المخزنة في العقد التي في الشجرة الفرعية اليمينية لـ v أكبر من k.

إن المفاتيح المخزنة في عقد T تتيح طريقة لتنفيذ البحث من خلال إجراء مقارنة عند كل عقدة داخلية v، ويمكن أن تتوقف عند v أو تتابع عند الابن اليساري أو اليميني لـ v. وبالتالي، فإن أشجار البحث الثنائية هي أشجار ثنائية مكتملة Complete or proper غير فارغة، وتستخدم العقد الخارجية كمقابض placeholders. إن هذا الأسلوب يبسط الكثير من خوارزميات البحث والتعديل الخاصة بها.

يمكن في بعض الأحيان أن نسمح باستخدام أشجار بحث ثنائية غير مكتملة improper، فهي تتيح استخداماً أفضل لمساحة الذاكرة، إلا أنها أكثر كلفة وتعقيداً في تحقيق خوارزميات البحث والتعديل.

إن الميزة الهامة لشجرة البحث الثنائية هي تحقيق قاموس مرتب. أي أن شجرة البحث الثنائية يجب أن تمثل أو تعبر بشكل شجري أو هرمي hierarchically عن ترتيب لمفاتيحها، باستخدام علاقة بين الأب والابن. وبتحديد أكبر، فإن التجول المرتب inoreder traversal لجميع عقد شجرة بحث ثنائية T يجب أن يقوم بزيارة جميع المفاتيح بالترتيب.

- خوارزمية اختبار فيما إذا كانت شجرة بحث ثنائي أو لا.

- If the current node is **null** then return **true.**
- If the value of the left child of the node is greater than or equal to the current node then return **false.**
- If the value of the right child of the node is less than or equal to the current node then return **false.**
- If the left subtree or the right subtree is not a BST then return **false.**
- Else return **true**

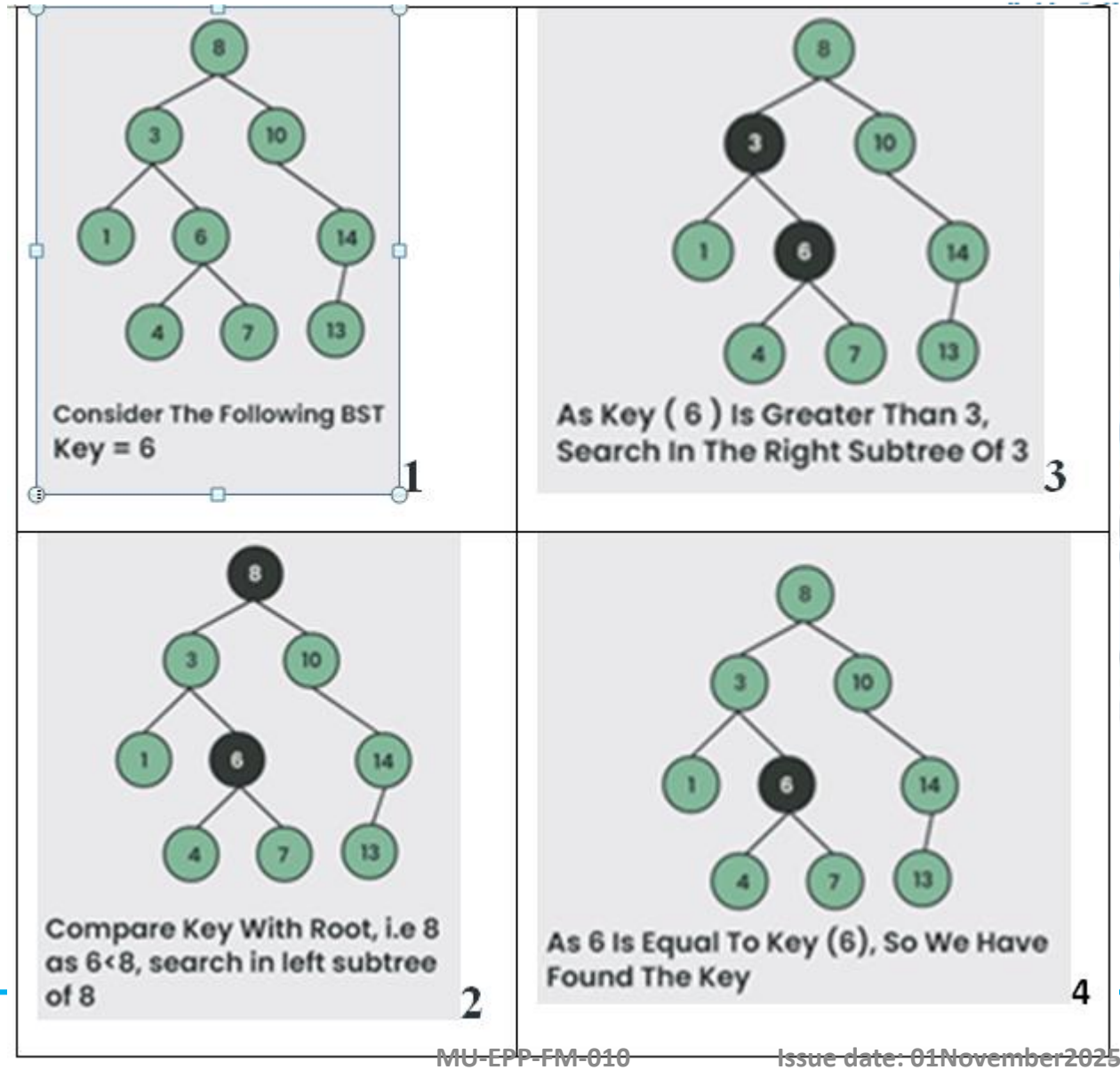
Basic Operations on BST

العمليات الأساسية في أشجار البحث الثنائية 1

Basic Operations on BST:

- Searching in Binary Search Tree
- Insertion in Binary Search Tree
- Deletion in Binary Search Tree
- Binary Search Tree (BST) Traversals – Inorder, Preorder, PostOrder

البحث في شجرة البحث الثنائية 1



لتنفيذ العملية find(k) في شجرة بحث ثنائية T، ننظر إلى الشجرة على أنها شجرة قرار decision tree، وتتم المقارنه عند كل عقدة داخلية v مع مفتاح البحث k وإذا كانت نتيجة مقارنته مع القيمة المخزنه في العقدة v، والمشار إليه بـ key(v).

- أصغر عندئذ يستمر البحث في الشجرة الفرعية اليسارية،
- أكبر عندئذ يستمر البحث في الشجرة الفرعية اليمينية.
- وإلا إنه (يساوي) عندئذ فإن البحث ينتهي بنجاح return true،
- إذا وصلنا إلى عقدة خارجية ولا تملك مفتاح البحث، فإن البحث ينتهي بالسلب ويعيد return false .

البحث في شجرة البحث الثنائية 2

يبين المقطع الخوارزمي 7-1 وصفاً مفصلاً لهذا الأسلوب، حيث k هو مفتاح البحث، وv هي عقدة من T.

تعيد الطريقة SearchTree عقدة (موقع) w من الشجرة الفرعية T(v) من الشجرة T والتي جذرها v، حيث يحصل أحد الأمور التالية:

- w هي عقدة داخلية ويملك العنصر w مفتاحاً مساوي لـ k.
- w هي عقدة خارجية تمثل موقع k في حالة تجول ترتيب عبر T(v)، إلا أن k ليست مفتاحاً متضمناً في T(v).

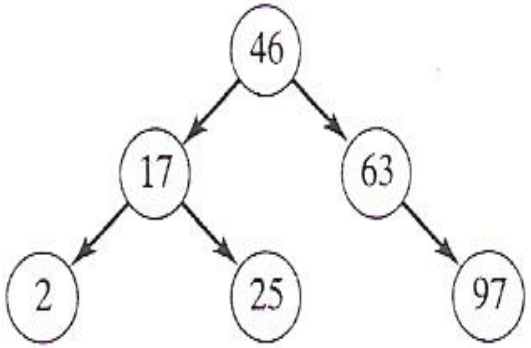
وبالتالي، يمكن إنجاز الطريقة find(k) من خلال استدعاء SearchTree (k,root()).

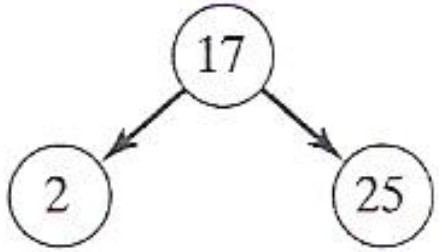
لتكن w هي العقدة من الشجرة T المعادة بهذا الاستدعاء. إذا كانت w هي عقدة داخلية، عندئذ نعيد قيمة w وإلا فإننا نعيد null.

ندرج خوارزمية البحث العودية

```
Algorithm SearchTree (k,v):
    if T.isExternal(v) then      return v
    if k<key(v) then             return SearchTree (k,T.left(v))
    else if k>key(v) then        return SearchTree (k,T.right(v))
    return v          {we know k=key(v)}
```

مثال 1عن البحث في شجرة البحث الثنائية 1





نفرض أننا نريد البحث ضمن الشجرة المجاورة عن القيمة 25. نبدأ بالجذر وبما أن 25 اصغر من القيمة في الجذر أي 46، فإننا نستنتج أن القيمة المرغوبة قد تكون موجودة في الجزء الذي على يسار الجذر وبالتالي تصبح العملية هي البحث ضمن الشجرة الفرعية اليسارية والتي جذرها هو 17.

نتابع الآن عملية البحث بمقارنة القيمة المراد البحث عنها 25 مع القيمة في جذر الشجرة الفرعية وبما أنها أكبر منها وبالتالي يجب أن يتم البحث في الجزء الذي على يمين العقدة الجذر أي:

باختبار القيمة في الجذر في الشجرة الفرعية المؤلفة من عقدة واحدة نصل إلى إيجاد العقدة المطلوبة.

وعندما يصل البحث إلى ورقة خارجية ولاتملك القيمة المبحوث عنها عندها نصل لنتيجة البحث السلبيه القيمة غير موجوده.

مثال 1عن البحث في شجرة البحث الثنائية 2

وبالتالي يجب إضافة التصريح عن التابع Search( )ضمن القسم public للصنف BST :

```
bool Search(const int  & item) const;
```

ومن كتابة الشيفرة التالية في الملف:

```
bool BST :: Search(const int  & item) const
{
BST * locptr = root;    bool found = false;
 for ( ; ; )
 {    if (found || locptr == 0) break;
  if (item < locptr->data)         // descend left
    locptr = locptr->left;
  else if (item > locptr->data)   // descend right
    locptr = locptr->right;
  else                found = true;   // item found
 }
 return found;
}
```

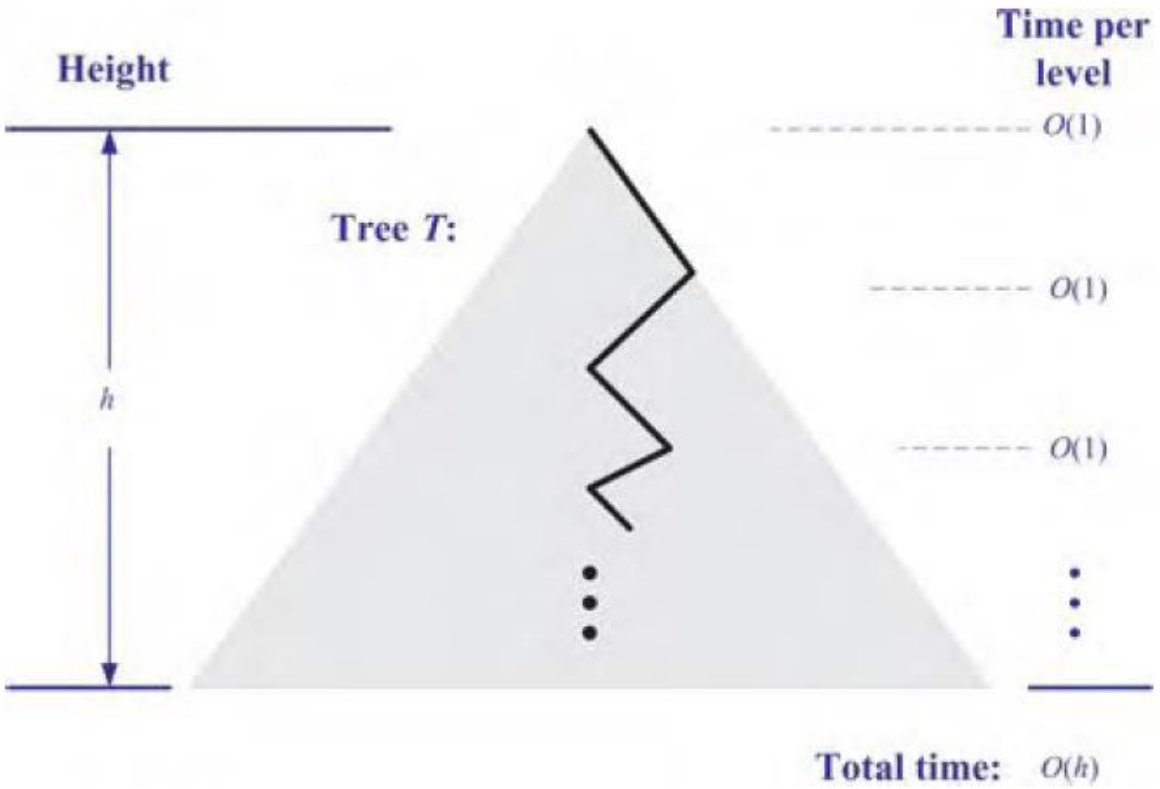
حيث يبدأ المؤشر locptr عند جذر شجرة البحث الثنائي ومن ثم ينتقل إلى الرابط اليميني أو اليساري للعقدة الحالية بحسب كون العنصر الذي نبحث عنه أصغر أو أكبر من القيمة المخزنة في العقدة. تستمر هذه العملية حتى يتم إيجاد العنصر المرغوب أو يصبح locptr صفرياً مشيراً إلى شجرة فرعية فارغة والتي تعني أن العنصر غير موجود في الشجرة.

تابع البحث في شجرة البحث الثنائية 1

```
// Utility function to search a key in a BST
BST BST::search(BST * root, int key)
{   // Base Cases: root is null or key is present at root
    if (root == NULL) return NULL;
    if (root->key == key)  return root ;
     // Key is greater than root's key
    if (root->key < key)return search(root->right, key);
     // Key is smaller than root's key
    return search(root->left, key);
}
```

Analysis of Binary Searching Tree

تحليل البحث في شجرة ثنائية 1



الشكل 7-2- كلفة البحث على شجرة ثنائية.

إن القيمه المقدمة الخوارزمية السابقة لتنفيذ العملية findAll(k) هى زمن O(h+s)، حيث s عدد القيم التي يتم إيجادها (إعادتها). إلا أن هذه الطريقة أكثر تعقيداً بقليل وستدرسها لاحقاً.

إن تحليل زمن تنفيذ الحالة الأسوأ لعملية البحث في شجرة بحث ثنائية T يعتبر أمراً بسيطاً. فالخوارزمية SearchTree هي خوارزمية عودية وتنفذ عدداً ثابتاً من العمليات البسيطة في كل عملية استدعاء عودي.

يتم تنفيذ كل استدعاء عودي للطريقة SearchTree على إبن من أبناء العقدة السابقة. أي أن، الطريقة SearchTree يتم استدعاؤها على عقد مسار من T يبدأ عند الجذر وينزل هبوطاً مستوى واحد في كل مرة. وبالتالي، عدد هذه العقد محدد بـ h+1، حيث h هو ارتفاع T.

بكلام آخر، بما أننا ننفق زمناً مقداره O(1) عند كل عقدة في البحث، فإن الطريقة find تنفذ في زمن O(h)، حيث h هو ارتفاع شجرة البحث الثنائية T المستخدمة لتحقيق الشجرة. (يبين الشكل 7-2 هذه العملية).

Update Operations(Insertion)

طرائق التعديل (الحشر) 1

تتيح أشجار البحث الثنائية تحقيقات للعمليات insert و remove باستخدام خوارزميات مباشرة تماماً.

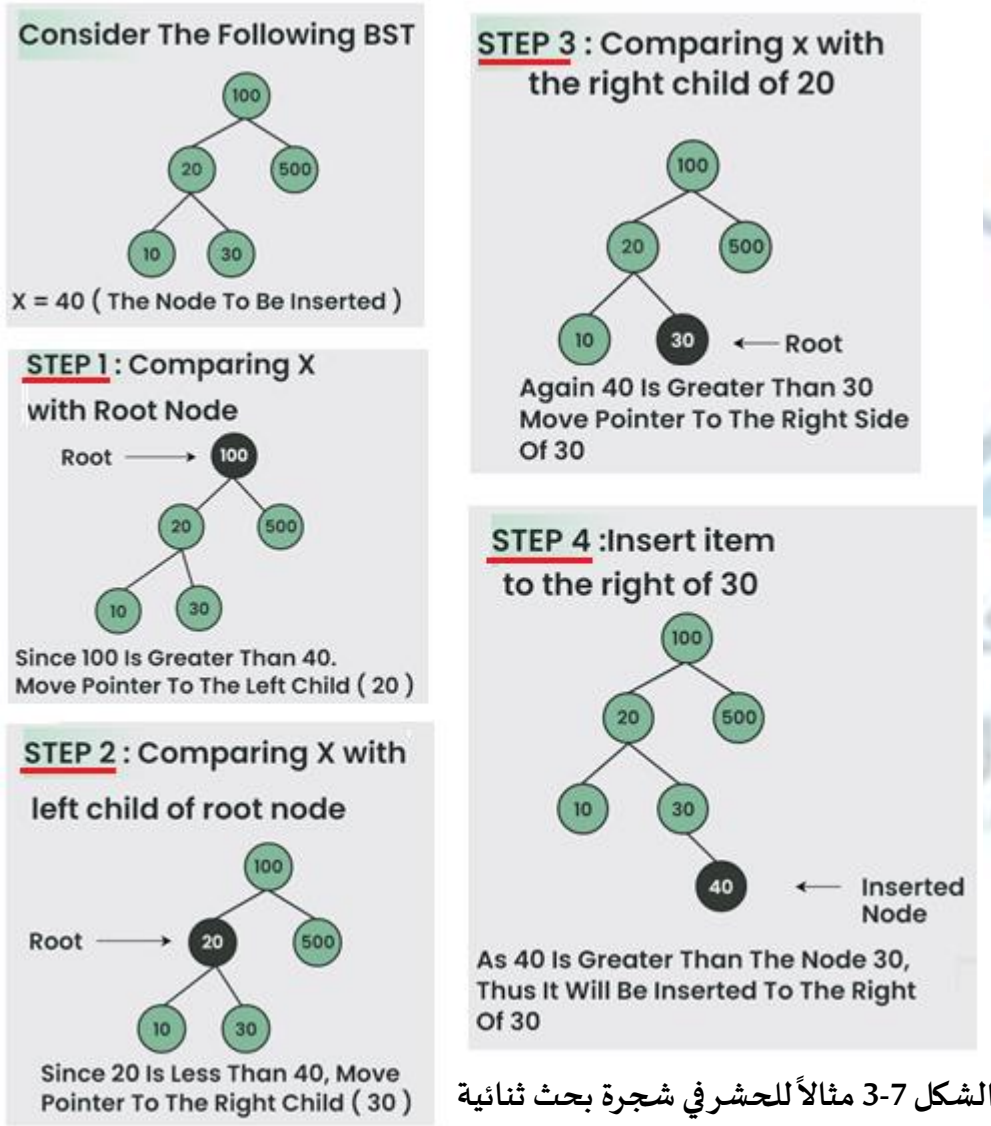
1- الحشر Insertion : لنفرض شجرة ثنائية T تتضمن عملية التعديل التالية:

- insertAtExternal(v,e): إضافة العنصر e في عقدة خارجية v وتوسيع v لتكون داخلية من خلال إعطائها المؤشرين الصادرين عنها القيمة NULL.

باستخدام هذه الطريقة، يمكن تنفيذ insert(k,x) والمبينة في المقطع 7-2.

```
BST* BST::Insert(BST* root, int value)
{if (!root) {return new BST(value);} // Insert the first node, if root is NULL.
if (value > root->data) {// Insert data.
   /*Insert right node data, if the 'value' to be inserted is greater than 'root' .Process right nodes.*/
root->right = Insert(root->right, value);}
else if (value < root->data) {
/* Insert left node data, if the 'value' to be inserted is smaller than 'root' node data. Process left nodes.*/
            root->left = Insert(root->left, value);}
            return root;} // Return 'root' node, after insertion
```

Update Operations(Insertion)



الشكل 7-3 مثالاً للحشر في شجرة بحث ثنائية

طرائق التعديل (الحشر) 2

تقوم هذه الخوارزمية بتتبع مسار بدءاً من جذر T إلى عقدة خارجية، والتي توسع إلى عقدة داخلية جديدة لاحتواء القيمة الجديدة. يبين الشكل 7-3 مثالاً للحشر في شجرة بحث ثنائية.

حشر عنصر قيمته 40 في شجرة البحث المبينة في الشكل 7-3.

(a) إيجاد الموقع

(b) الشجرة الناتجة.

طرائق التعديل (الحذف) 1

Update Operations Delete

الحذف Delete

يعتبر تحقيق العملية delete(k) شجرة بحث ثنائية T أمراً أكثر تعقيداً بقليل من عمليات الحشر، وذلك لأننا لانرغب بإنشاء أي ثقوب أو فجوات في الشجرة T. نفرض في هذه الحالة، أن أي شجرة ثنائية تدعم عملية التحديث الإضافية التالية:

- نبدأ تحقيقنا للعملية delete(k) كنمط بيانات مجرد باستدعاء SearchTree (k,T.root()) على T لإيجاد العقدة من T التي تخزن عنصراً ذو مفتاح يساوي k هناك احتمالين.

- الأول:أن نصل لعقدة خارجية ولم نجد القيمة المبحوث عنها عندها تعيد الخوارزمية SearchTree القيمة null و لن يتم الحذف.

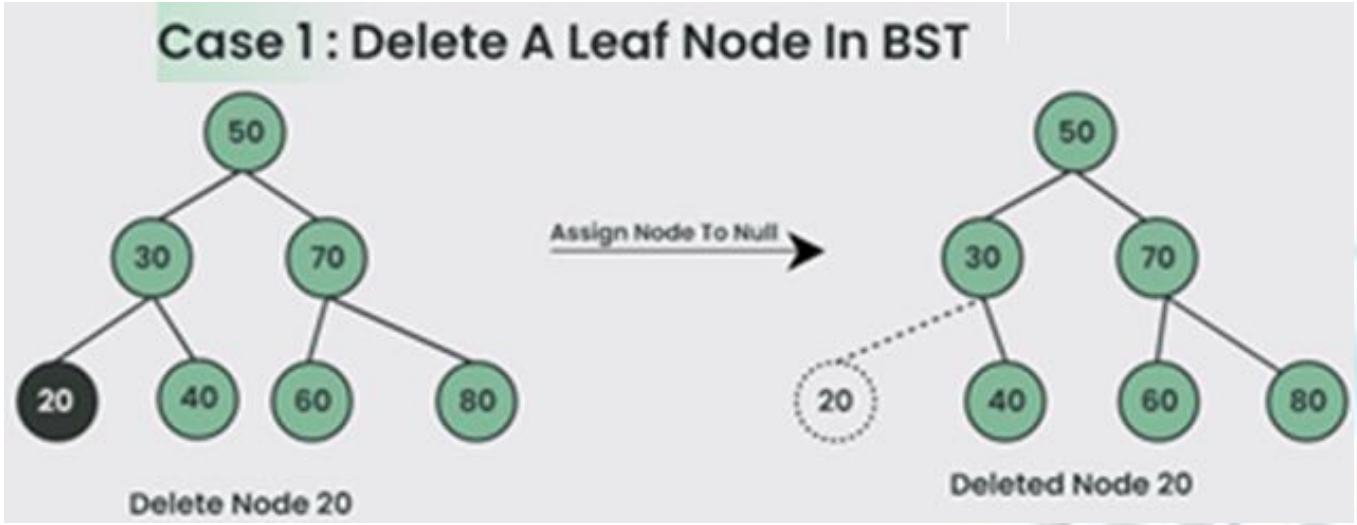
- الثاني: العنصر موجود وهناك ثلاث حالات أن تكون ورقه، أو لها ابن وحيد أو لها ابنان:

1-2 - إذا أعاد SearchTree عقدة خارجية (ورقة)، عندها يتم وضع المؤشر الذي يشير إليها يساوي null ونعيد العقدة المحذوفة لخزان العقد الفارغة.

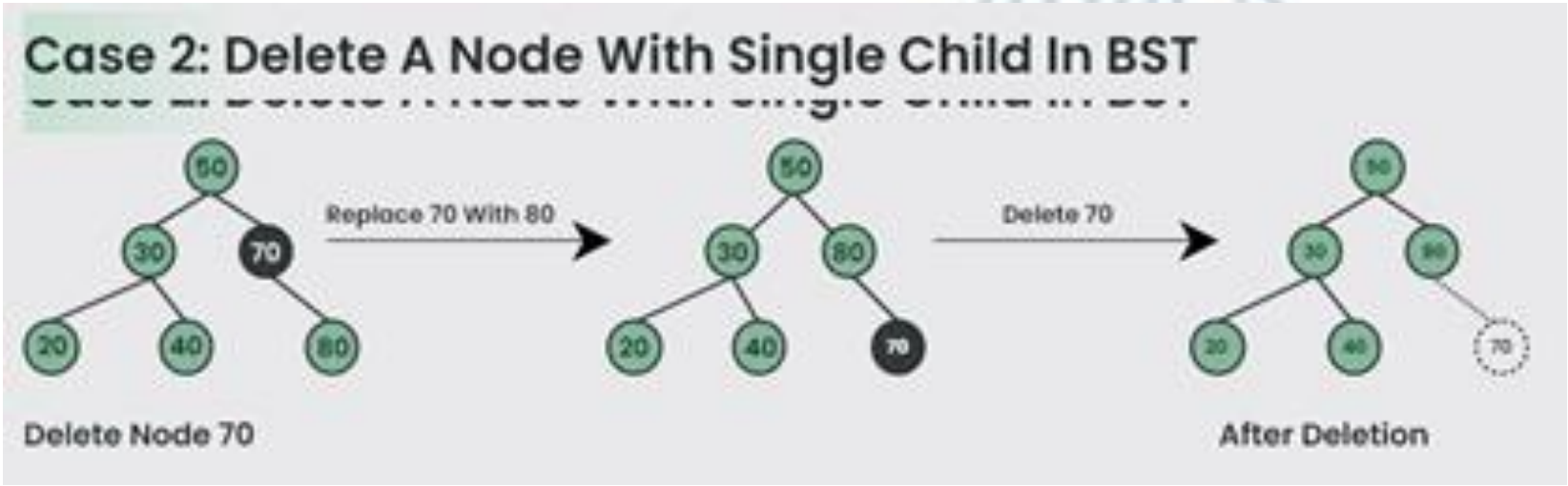
2-2- أما إذا أعاد SearchTree عقدة داخلية w ولها إبن وحيد t، عندئذ نبادل محتوى العقدة wمع ابنها الوحيد t ونقوم بحذف الابن t ونعيد العقدة المحذوفة لخزان العقد الفارغة. أي نقوم بإعادة تشكيل T من خلال استبدال wبإبنها الوحيد (الشكل 7-4).

طرائق التعديل (الحذف) 2

Delete الحذف



1-2 - إذا أعاد SearchTree عقدة خارجية (ورقة)، عندها يتم وضع المؤشر الذي يشير إليها يساوي null ونعيد العقدة المحذوفة لخزان العقد الفارغة.



1-2 - إذا أعاد SearchTree عقدة داخلية ولها ابن وحيد، عندها يتم وضع المؤشر الذي يشير إليها يشير إلى ابنها ونعيد العقدة المرغوب حذفها لخزان العقد الفارغة.

الحذف Delete

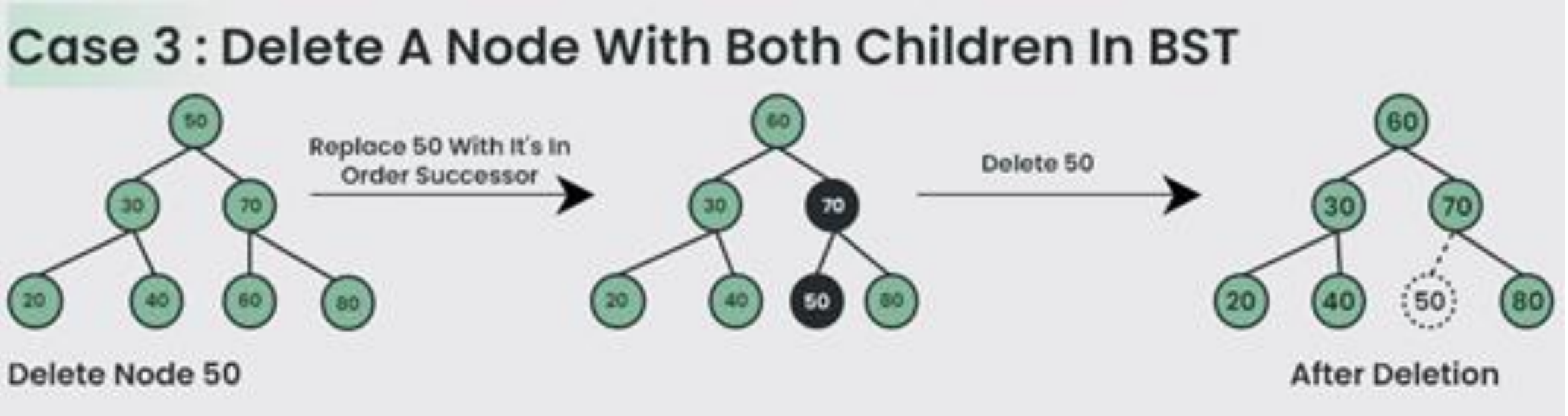
3-2- إذا كانت العقدة المرغوب حذفها w تملك ابنين، لانستطيع ببساطة حذف العقدة w من T لأن ذلك سيؤدي إلى إنشاء فجوة أو ثقب في T. بدلاً من ذلك فإننا نقوم بمايلي (الشكل 7-5):

- نقوم بإيجاد العقدة الخلفSuccessor للعقدة المحذوفة وتكون أول عقدة داخلية مثل y تلي w في تجول مرتب عبر T. تكون العقدة y هي العقدة التي تقع في أقصى اليسار من شجرة الفرعية اليمينية لـ w، ويتم إيجادها من خلال الذهاب إلى الابن اليميني لـ w أولاً ومن ثم هبوط T باتجاه اليسار،(عندما نجد عقده ابنها اليساري NULL تكون الخلف.بتتبع الأبناء اليساريين.
- نقوم بحفظ العنصر المخزن في w في متحول مؤقت t، ونسخ القيمة المخزنه في y إلى w. إن هذا التصرف له مفعول حذف العنصر السابق المخزن في w.
- نقوم بحذف العقد y من خلال استدعاء تابع الحذف على T. هذا التصرف يستبدل y بأبنها اليميني إن كان لها أبن يمين.
- نقوم بإعادة العنصر المخزن سابقاً في w والذي قمنا بحفظه في المتحول المؤقت t.

وكما في حالتي البحث والحشر، فإن خوارزمية الحذف تتجول عبر مسار من الجذر إلى عقدة خارجية، وتقوم ربما بنقل عنصر بين عقدتين من هذا المسار، ومن ثم تنفذ عملية removeExternalعند تلك العقدة الخارجية.

ملاحظة: قد تجد في بعض المراجع إستبدال العقدة المحذوفة بأكبر قيمة في الفرع اليساري وبمحاكات مشابهه يمكن الوصول لها.

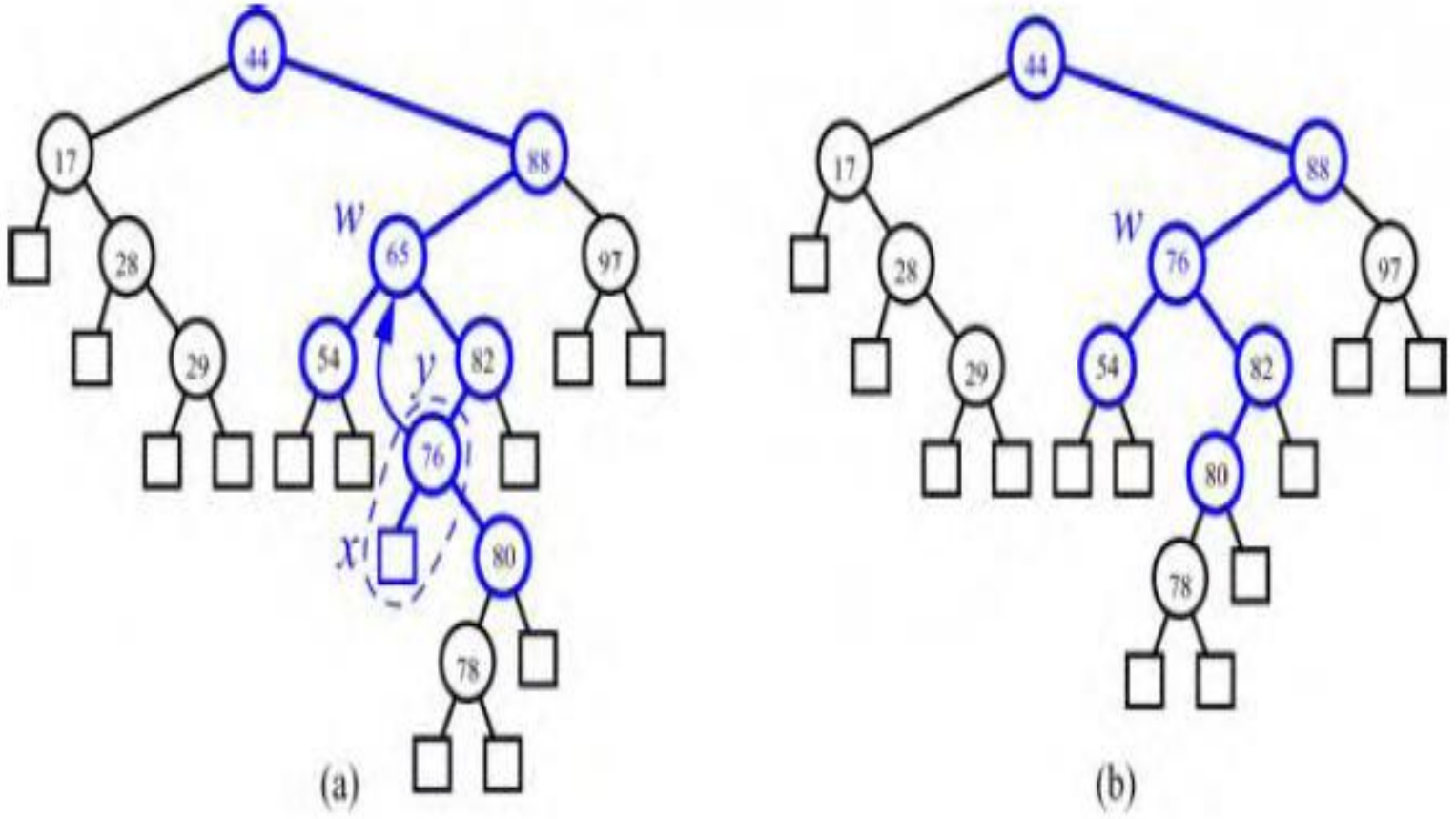
Delete الحذف



2-2- العنصر المراد حذفه (50) مخزن في الجزر وهى عقدة w ونشير لها ب t ولها إبنين لذلك نبحث عن Successor وهو العنصر في اقصى يسار الفرع اليميني ويتحقق بلإتجاه أول مرة لليمين ونبحث عن العنصر الذي مؤشره اليساري NULL مستخدمين المؤشر t، وفي حالتنا 60 وهى عقدة خارجيه وهنا نشير لها ب y وننسخ محتواها 60إلى w والقيمة 50 إليها ونقوم بحذف y وكذلك المؤشر t (أي البحث عن الخلف y ومبادلة قيمة w بقيمة y بمساعدة t، وحذف كل من المتبدل به y والمساعد t).

طرائق التعديل (الحذف) 5

Removal الحذف



تابع إيجاد أصغر العناصر:

```
Node *findmin( Node *r)
{
   if (r==NULL) return NULL;
   else if (r->left==NULL)return r;
   else return findmin(r->left);
}
```

الشكل 7-5- الحذف من شجرة البحث الثنائية المبينة في الشكل 7-3 حيث العنصر المراد حذفه (ذو المفتاح 65) مخزن في عقدة w كلا إبنيها داخليين(a قبل، b بعد).

Binary Search Tree (BST) Traversals – Inorder, Preorder, Postorder

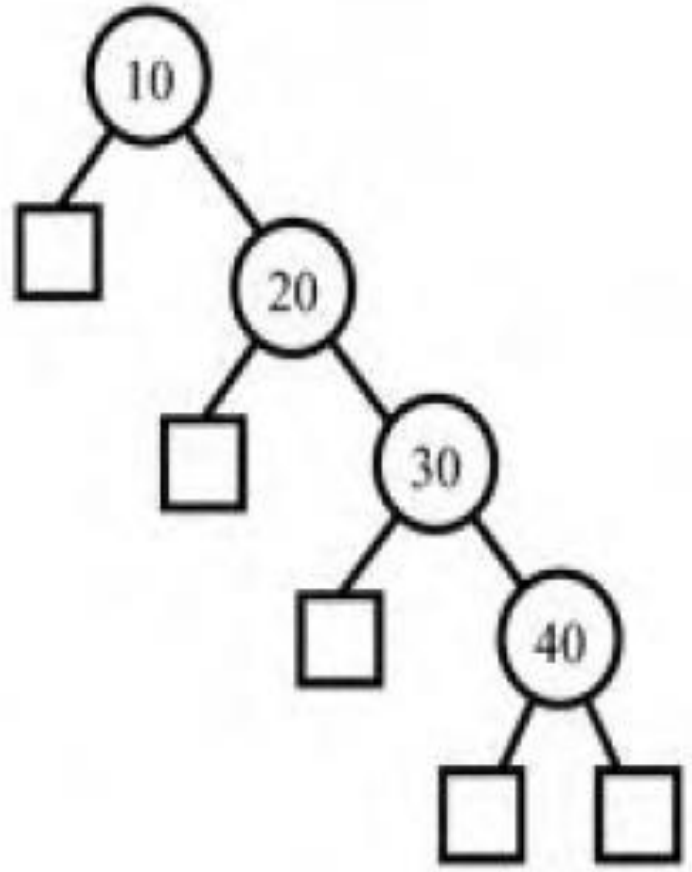
Preorder (PLR) *At first visit the **root** then traverse **left subtree** and then traverse the **right subtree**.*

Postorder (LRP) :*At first traverse **left subtree** then traverse the **right subtree** and then visit the **root**.*

Inorder (LPR) : *At first traverse **left subtree** then visit the **root** and then traverse the **right subtree**.*

Performance of a Binary Search Tree

أداء شجرة بحث ثنائية

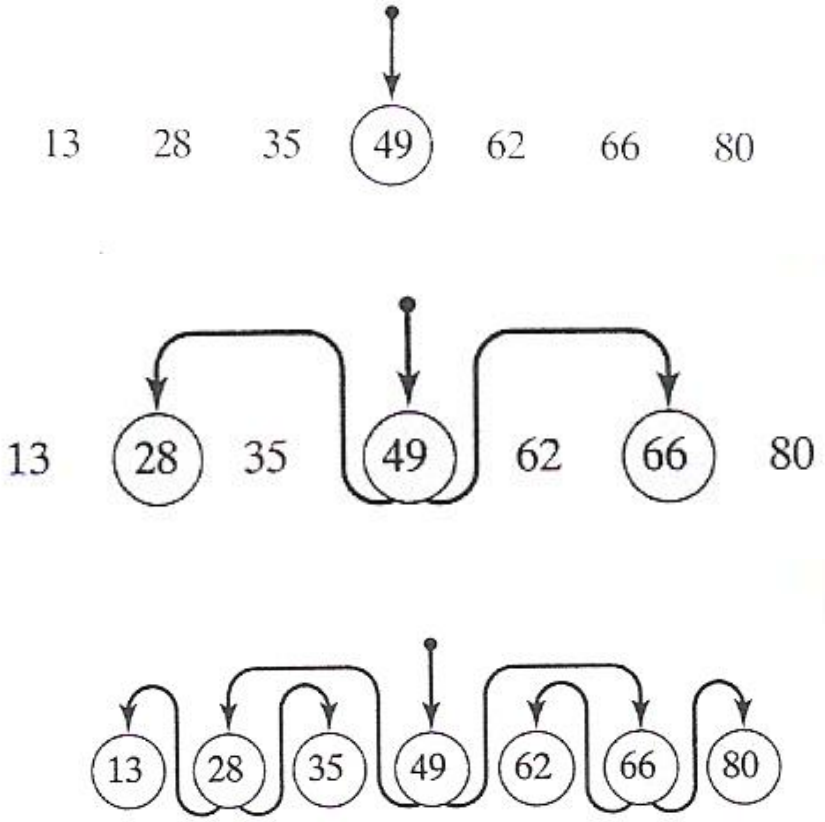


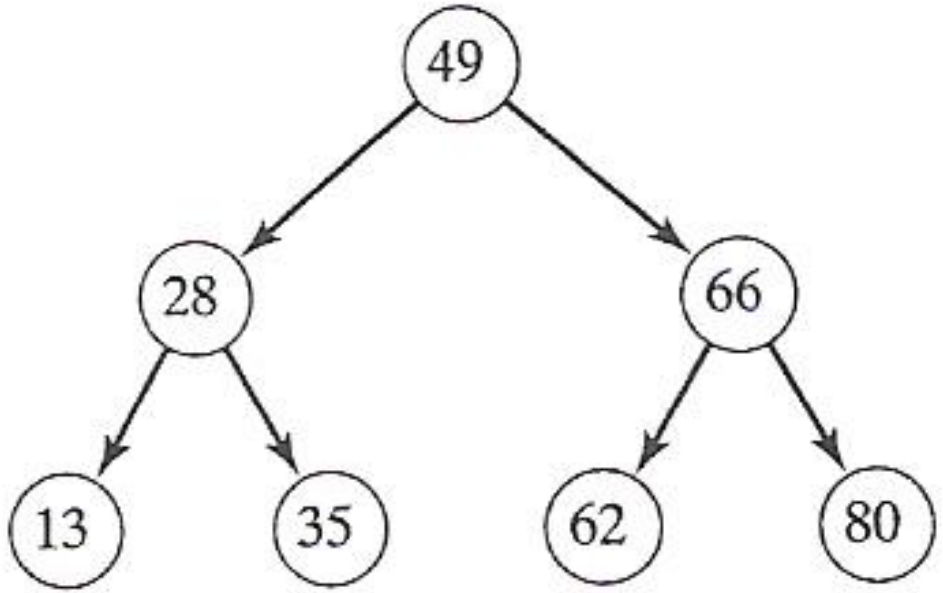
الشكل 7-6- مثال لشجرة ثنائية ذات ارتفاع خطي.

إن تحليل خوارزميات البحث، الحشر والحذف متشابه. نحن ننفق زمناً O(1) عند كل عقدة نزورها، وفي الحالة الأسوأ، عدد العقد التي تتم زيارتها تتناسب مع الارتفاع h للشجرة T. وبالتالي، في قاموس D تم تحقيقه بشجرة بحث ثنائية T فإن الطرائق find، insert، و remove تنفذ في زمن O(h)، حيث h هو ارتفاع T. وبالتالي، إن شجرة البحث الثنائية T هي تحقيق فعال أو مجدي لقاموس ذي n عنصر فقط إذا كان ارتفاع الشجرة صغيراً. في الحالة الأفضل، يكون ارتفاع T هو h=log(n+1) الأمر الذي ينتج أداء ذو زمن لوغاريتمي لجميع عمليات القاموس. في حين أنه، في الحالة الأسوأ، يكون ارتفاع T هو n، وفي هذه الحالة قد يكون أفضل أن نستخدم تحقيقاً للقاموس على شكل لائحة مرتبة.

تحدث الحالة الأسوأ عندما نقوم بإدخال سلسلة من العناصر ذات مفاتيح بترتيب متزايد أو متناقص (الشكل 7-6).

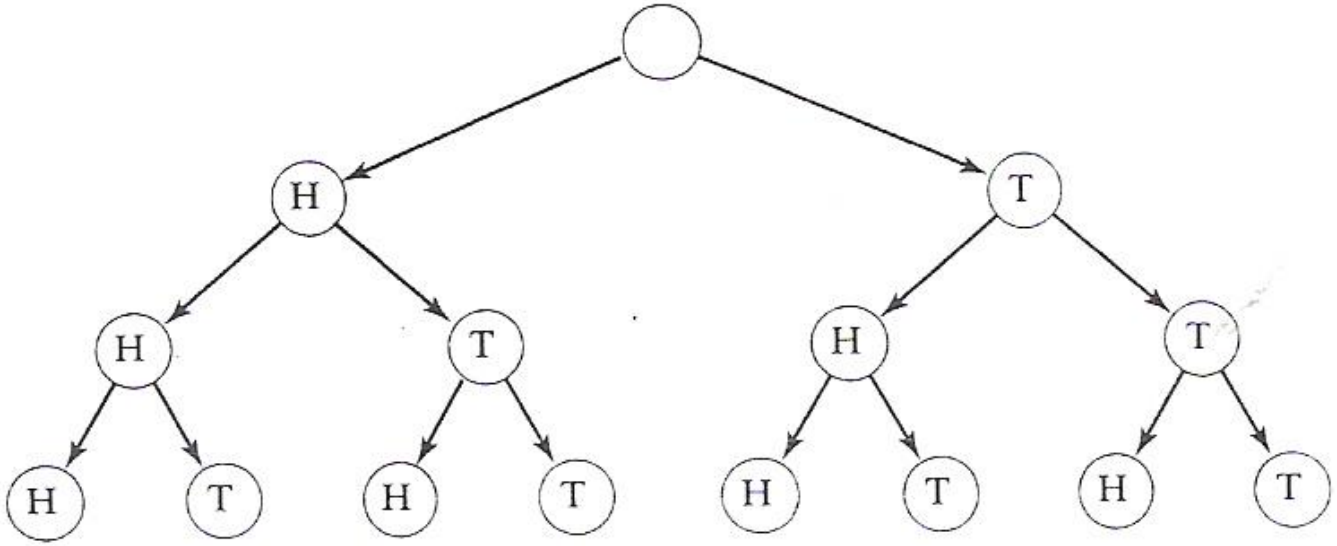
إن تمثيل لائحة مرتبة كشجرة ثنائية يذكرنا بعملية البحث الثنائي جيث أن mad تمثل العقد على أن يتم التجوال بكلا الجزئين.





الشكل 7-7- يمثل شجرة البحث الثنائي.

شجرة ثنائية لتمثيل الاحتمالات الممكنة لعملية رمي قطعة نقدية ثلاث مرات (وتطبق على التجارب والاختبارات التي يكون لها احتمالين ممكنين (مثل off أو on، 0 أو 1، false أو true، down أو up).

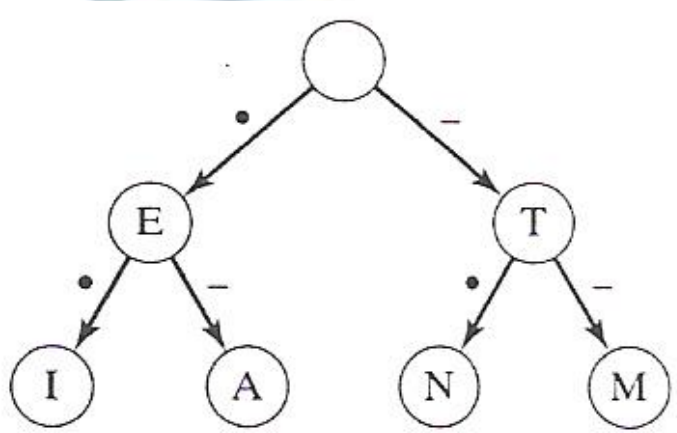


يمثل كل مسار بدءاً من الجذر وحتى إحدى الأوراق ناتجاً محتملاً مثل THT (نقش ثم طرة ثم نقش) كما هو مبين في المخطط.

الشكل 7-8- يمثل شجرة البحث الثنائي رمي قطعة نقدية ثلاث مرات.

في مسائل التشفير مثل تشفير و فك تشفير coding and decoding الرسائل المرسلة باستخدام شيفرة مورس Morse code، وفي هذه الطريقة يتم تمثيل المحارف كتتال من النقط والشرطات كما هو مبين في الجدول التالي:

A ·—	M ——	Y —·——
B —···	N —·	Z ——··
C —·—·	O ———	1 ·————
D —··	P ·——·	2 ··———
E ·	Q ——·—	3 ···——
F ··—·	R ·—·	4 ····—
G ——·	S ···	5 ·····
H ····	T —	6 —····
I ··	U ··—	7 ——···
J ·———	V ···—	8 ———··
K —·—	W ·——	9 ————·
L ·—··	X —··—	0 —————



الشكل 7-9- يمثل جزء من شيفرة مورس .

تستخدم العقد في الشجرة الثنائية لتمثيل المحارف وكل قوس من عقدة إلى أبنائها تميز إما بنقطة أو شرطة بحسب فيما إذا كانت تقود إلى ابن يميني أو ابن يساري على التوالي والتالي فإن أجزاء الشجرة الخاصة بشيفرة مورس هي:
تتالي النقط والشرطات المميز للمسار من الجذر إلى عقدة ما يمثل شيفرة مورس لذلك الحرف

الملف الرأسي BST.h:

```
// C++ program to demonstrate insertion
// in a BST recursively
#include <iostream>
using namespace std;
class BST {
public:  int data;      BST *left, *right;

BST();                          // Default constructor.
BST(int);               // Parameterized constructor.
BST* Insert(BST*, int);         // Insert function.
void Inorder(BST*); // Inorder traversal.
void Postorder(BST*);           // Postorder traversal
void Preorder(BST*);            // Preorder traversal.
BST* deleteNode(BST*, int); // Delete function.
};
```

ملف التوابع BSTF.CPP

```
#include <iostream>
#include "BST.h"
using namespace std;
// Default Constructor definition.
BST::BST(): data(0) , left(NULL), right(NULL){ }
// Parameterized Constructor definition.
BST::BST(int value){data = value; left = right = NULL;}
// Insert function definition.
BST* BST::Insert(BST* root, int value)
{if (!root) {// Insert the first node, if root is NULL.
return new BST(value);}
// Insert data.
if (value > root->data) {root->right = Insert(root->right, value);}
   /*Insert right node data, if the 'value‘ to be inserted is greater than 'root' .Process right nodes.*/
```

```
else if (value < root->data) {
/* Insert left node data, if the 'value' to be inserted is smaller than 'root' node data. Process left nodes.*/
        root->left = Insert(root->left, value);}


    return root;} // Return 'root' node, after insertion

// Preorder, Postorder, Inorder traversal function.This gives data in sorted order.

void BST::Preorder(BST* r) // root ->left->right
   { if (r == NULL) return;      cout << r->data << "\t";        Preorder(r->left);            Preorder(r->right); }

void BST::Postorder(BST* r) // left--> right->root
{ if (r == NULL) return;         Postorder(r->left);            Postorder(r->right);          cout << r->data <<"\t";}

void BST::Inorder(BST* root) // left--> root-> right
 {if (root == NULL) {return;}Inorder(root->left);          cout << root->data << " ";     Inorder(root->right);}
```

ملف التوابع BSTF.CPP

```
// Function that returns the node with minimum  key value found in that tree
BST * minValueNode(BST * node)
{   BST * current = node;// Loop down to find the leftmost leaf
    while (current && current->left != NULL)     current = current->left;
            return current;}
// Function that deletes the key and returns the new root
BST* BST::deleteNode(BST* root, int value)
   {   // base Case
      if (root == NULL)       return root;
  // If the key to be deleted is smaller than the root's key,
  // then it lies in left subtree
    if (value < root->data)
            {root->left = deleteNode(root->left, value);}
 // If the key to be deleted is greater than the root's key,
```

```
// then it lies in right subtree
else if (value > root->data)
            { root->right= deleteNode(root->right, value);}
// If key is same as root's key, then this is the node to be del
else { // Node with only one child or no child
   if (root->left == NULL)                  // one child on the right
            { BST * temp = root->right;      free(root);    return temp;}
else if (root->right == NULL)               // one child on the left
            { BST * temp = root->left;       free(root);    return temp; }
 // Node with two children:Get the inorder successor(smallest in the right subtree)
BST * temp = minValueNode(root->right);

// Copy the inorder successor's content to this node
root->data = temp->data;
```

ملف التابع الرئيس MAIN.CPP

```
// Delete the inorder successor
            root->right= deleteNode(root->right, temp->data);    }
    return root;
}
#include "BST.h"
using namespace std;
// Driver code
int main(){
  /* Let us create following BST
          50
       /     \
      30      70
     / \     / \
    20  40  60  80 */
```

ملف التابع الرئيس MAIN.CPP

```
BST b, *root = NULL;          root = b.Insert(root, 50);          b.Insert(root, 30);
b.Insert(root, 20);           b.Insert(root, 40);                 b.Insert(root, 70);
b.Insert(root, 60);           b.Insert(root, 80);
printf("\n Original BST in Preorder:\t");          b.Preorder(root);
printf("\n Original BST in Postorder:\t");         b.Postorder(root);
printf("\n Original BST: ");                       b.Inorder(root);
printf("\n\nDelete a Leaf Node: 20\n");           root = b.deleteNode(root, 20);
printf("Modified BST tree after deleting Leaf Node:\n");                    b.Inorder(root);
printf("\n\nDelete Node with single child: 70\n");                        root = b.deleteNode(root, 70);
printf("Modified BST tree after deleting single child Node:\n");           b.Inorder(root);
printf("\n\nDelete Node with both child: 50\n");                          root = b.deleteNode(root, 50);
printf("Modified BST tree after deleting both child Node:\n");             b.Inorder(root);
system("pause");              return 0;
}
```

ملف نتائج التنفيذ

Original BST in Preorder: 50 30 20 40 70 60 80

Original BST in Postorder: 20 40 30 60 80 70 50

Original BST in Inorder: 20 30 40 50 60 70 80

Delete a Leaf Node: 20

Modified BST tree after deleting Leaf Node:

30 40 50 60 70 80

Delete Node with single child: 70

Modified BST tree after deleting single child Node:

30 40 50 60 80

Delete Node with both child: 50

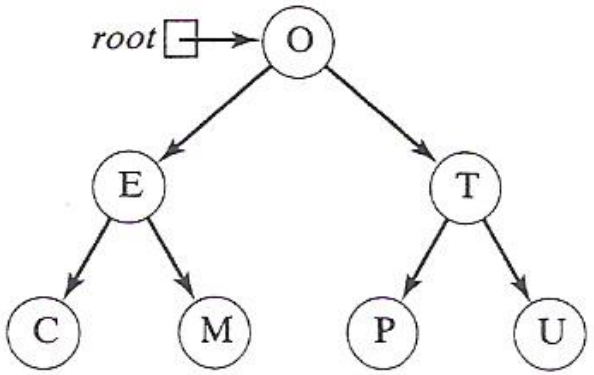
Modified BST tree after deleting both child Node:

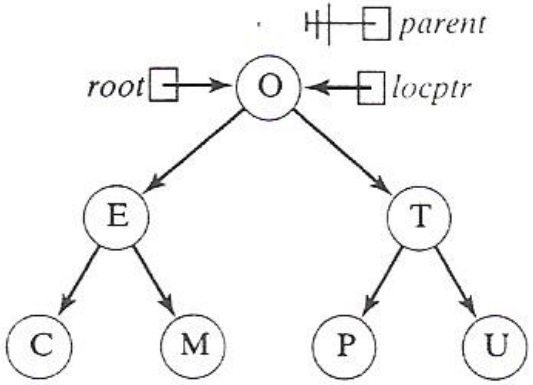
30 40 60 80

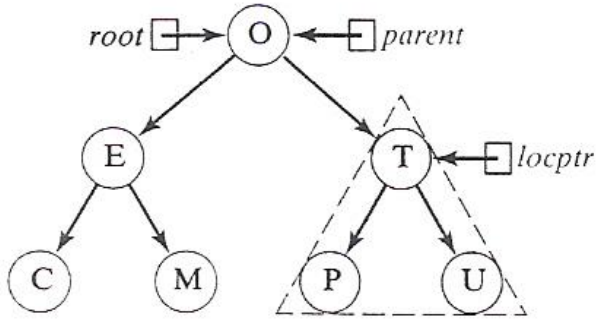
Press any key to continue . . .

انتهت محاضرة الأسبوع 9

مثال 2 عن الحشر1







يمكن بناء شجرة البحث الثنائية بالاستدعاء المتكرر لتابع يقوم بحشر العناصر في شجرة البحث الثنائية التي هي فارغة في الحالة الابتدائية (root يشير إلى المؤشر الصفري). الطريقة المستخدمة لتحديد المكان الذي سيتم حشر العنصر فيه مشابهة لتلك المستخدمة في عملية البحث، وبالتالي نحتاج فقط لأن نعدل التابع ( )Search للمحافظة على مؤشر إلى العقدة الأب للعقدة المختبرة حالياً عندما ننزل عبر الشجرة باحثين عن مكان لحشر العنصر.

لتوضيح ذلك، نفرض أن شجرة البحث الثنائية المجاوره تم بناؤها:

و أننا نريد إضافة الحرف 'R'. نبدأ عند الجذر ونقارن 'R' مع الحرف الموجود ضمنها:

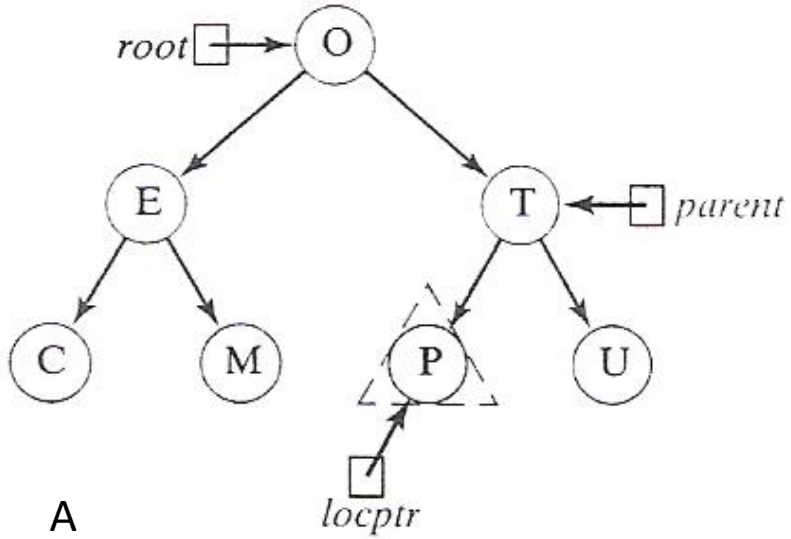
بما أن 'R'>'O' ننزل إلى الشجرة الفرعية اليمنى:

وبعد مقارنة 'R' مع 'T' المخزن في جذر هذه الشجرة الجزئية المشار إليها بواسطة locptr، نهبط إلى الشجرة الجزئية اليسرى بما أن 'R'<'T':

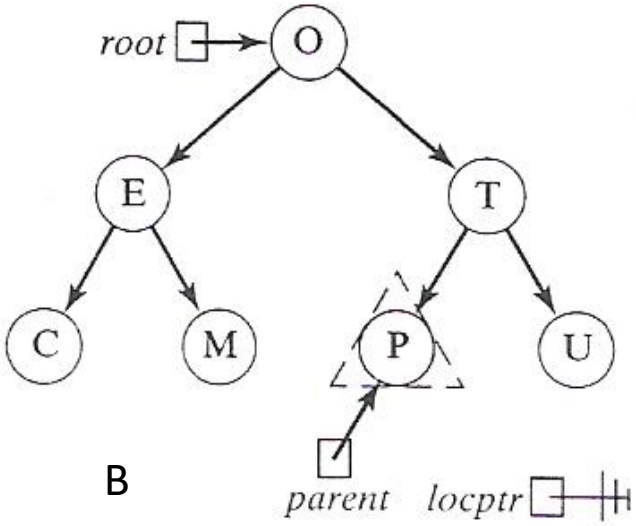
مثال2 عن الحشر2

وبما أن 'R'>'P' ننزل إلى الشجرة الجزئية اليمنى من هذه الشجرة الجزئية ذات العقدة الوحيدة الحاوية على 'P':

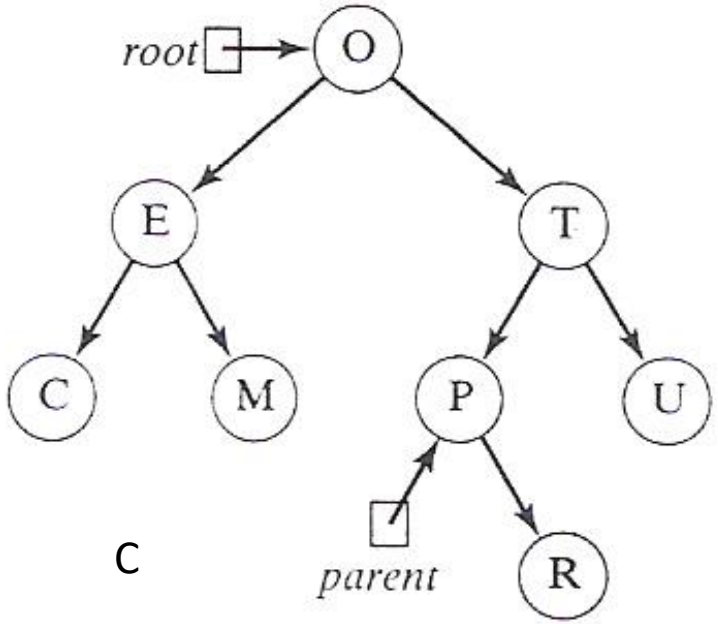
إن حقيقة أن الشجرة الجزئية اليمنى فارغة (أي locptr==null) تعني أن 'R' ليست موجودة في شجرة البحث الثنائية ويجب أن تحشر كابن يميني لهذه لعقدة الأب 'P'.



A



B



C

تطبيق على الأشجار الثنائية: شيفرة هوفمان1

تستخدم الأشجار الثنائية في العديد من مسائل التشفير وفك التشفير.

- شيفرة مورس، التي تمثل كل حرف بتتال من النقط والشرطات. تستخدم شيفرة مورس متتاليات متغيرة الطول.
- ترميزات الـ ASCII، EBCDIC، UNICODE التي يكون فيها طول الرمز هو نفسه لكل المحارف.
- شيفرة هوفمان Huffman code والتي تستخدم رموزاً بأطوال متغيرة.

الفكرة الأساسية في طرق التشفير متغير الطول هي استخدام شيفرات أقصر للأحرف الأكثر استخدماً، وشيفرات أطول للأحرف الأقل استخداماً. على سبيل المثال 'E' في شيفرة مورس هي عبارة عن نقطة واحدة، في حين أن 'Z' ممثلة بالشكل (.- -.).

إن الهدف هو تقليل الطول المتوقع لشيفرة الحرف، هذا يقلل عدد البتات الواجب إرسالها عند نقل الرسائل المشفرة. إن هذه الطرق في الترميز المتغيرة الطول مفيدة عند ضغط البيانات لأنها تقلل عدد البتات الواجب تخزينها

لتوضيح المسألة بشكل أدق، نفرض أننا أعطينا مجموعة المحارف $\{C_1,C_2,\ldots,C_n\}$ وأن أوزاناً محددة أرفقت مع هذه المحارف $w_1,w_2,\ldots,w_n$، حيث w_i هي الوزن المرفق بالمحرف C_i وهو يدل على تكرار استخدام هذا المحرف في الرسائل المراد تشفيرها. إذا كانت $l_1,l_2,\ldots,l_n$ هي أطوال الشيفرات الخاصة بالأحرف $C_1,C_2,\ldots,C_n$ على التوالي، عندئذ فإن الطول المتوقع لشيفرة أي من الأحرف تحسب كما يلي:

$$\exp ectedlength = w_1l_1 + w_2l_2 + \ldots..+ w_nl_n = \sum_{i=1}^{n} w_il_i$$

تطبيق على الأشجار الثنائية: شيفرة هوفمان2

كمثال بسيط، لندرس الأحرف الخمسة A,B,C,D,E ونفرض أنها تحصل بالأوزان التالية:

character	A	B	C	D	E
weight	0.2	0.1	0.1	0.15	0.45

في شيفرة مورس إذا استبدلنا النقطة بصفر والشرطة بواحد فإن هذه المحارف مشفرة كما يلي:

character	A	B	C	D	E
code	01	1000	1010	100	0

وبالتالي فإن الطول المتوقع لكل من هذه الأحرف الخمسة في هذه الطريقة هو:

$$0.2\times2+0.1\times4+0.1\times4+0.15\times3+0.45\times1=2.1$$

قابلية فك التشفير مباشرة immediate decidability:

الميزة المفيدة الأخرى لبعض طرق التشفير هو أنها قابلة لفك التشفير مباشرة immediately decodable. وهذا يعني ليس هناك تتالي من البتات يمثل محرفاً يمكن أن يكون جزءاً سابقاً prefixمن تتالي أطول لأحرف أخرى. وبالتالي عند استقبال تتالي من البتات فهو شيفرة لحرف، فهو يمكن أن يفك تشفيره إلى ذلك المحرف مباشرة بدون انتظار فيما إذا كانت الخانات التي تلي تجعل منه شيفرة لمحرف آخر.

لاحظ أن شيفرة مورس السابقة ليست قابلة لفك التشفير مباشرة، لأنه، وعلى سبيل المثال، الشيفرة للمحرف E هي (0) وهي جزء سابق من شيفرة الحرف A أي (01)، وكذلك شيفرة الحرف D أي (100) هي جزء سابق من شيفرة الحرف B أي (1000).
تستخدم شيفرة مورس من أجل عملية فك التشفير خانة إضافية هي الفراغ للفصل بين المحارف.

شيفرة هوفمان Huffman codes:

يمكن استخدام الخوارزمية التالية المقدمة من قبل D.A.Huffman في عام 1952 لبناء طريقة تشفير قابلة للفك مباشرة وتملك طولاً أصغرياً متوقعاً لشيفرة كل حرف:

HUFFMAN'S ALGORITHM

1-initialize a list of one-node binary trees containing the weights $w_1, w_2, \ldots, w_n$ one for each of the characters $C_1, C_2, \ldots, C_n$.

2-do the following n-1 times:

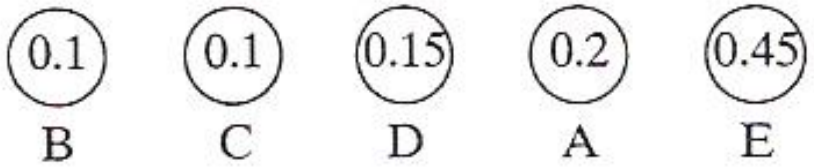
a-find two trees T' and T'' in this list with roots of minimal weights w' and w''.

b-replace these two trees with a binary tree whose root is $w'+w''$, and whose subtrees are T' and T'', and label the pointers to these subtrees 0 and 1 respectively.

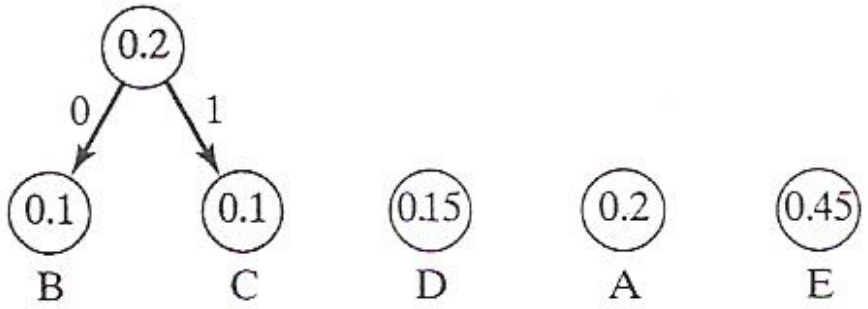
3-the code for character C_i is the bit string labeling a path in the final binary tree from the root to the leaf for C_i.

تطبيق على الأشجار الثنائية: شيفرة هوفمان4

كتوضيح لخوارزمية هوفمان، ندرس مجدداً المحارف A,B,C,D,E بالأوزان المحددة سابقاً. نبدأ ببناء لائحة من أشجار ثنائية مؤلفة من عقدة واحدة لكل محرف:



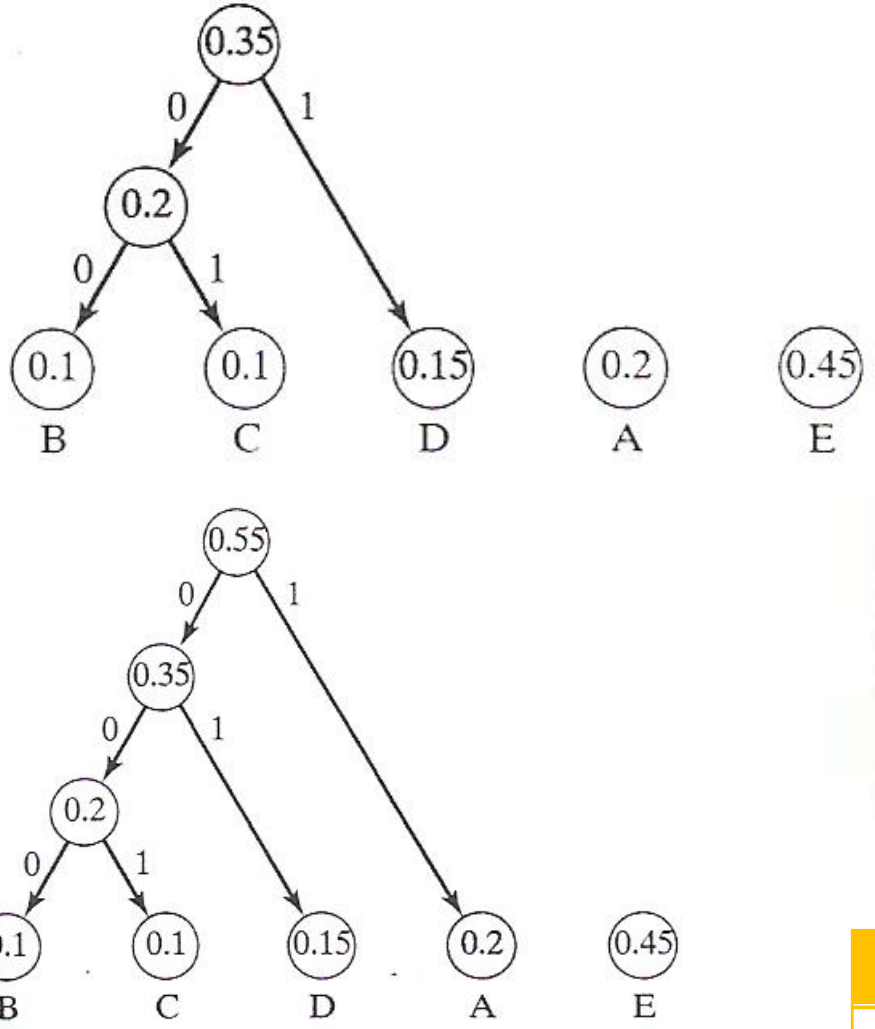
لشجرتين الأولى والثانية التي سيتم اختيارهما هي تلك الممثلة للمحارف B و C وذلك لأنها تملك الأوزان الأصغر. وبتجميع هاتين الشجرتين نشكل شجرة تملك الوزن 0.2=0.1+0.1 ولها شجرتين كشجرتين جزئيتين:



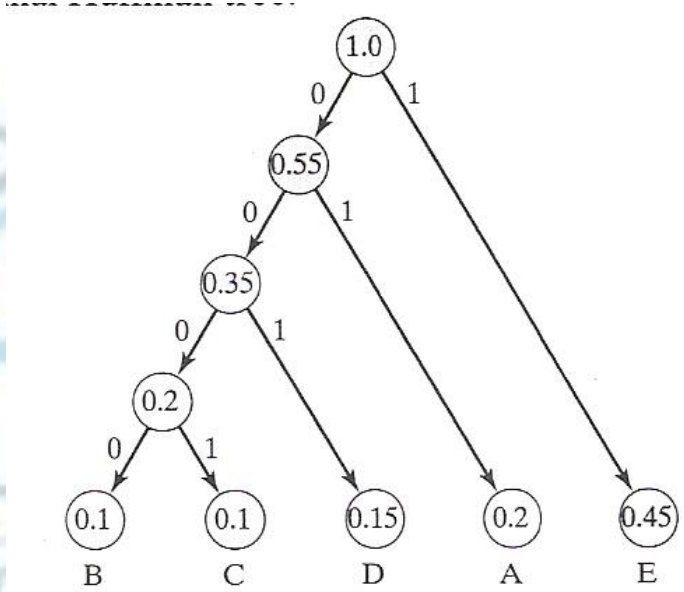
نختار من جديد من لائحة الأشجار الأربعة وزنين أصغريين، الأولى والثانية (أو الثانية والثالثة) ونقوم باستبدالهما كما في المرة السابقة بشجرة أخرى وزنها 0.35=0.15+0.2 ولها هاتين الشجرتين كشجرتين جزئيتين:

application of binary trees: Huffman codes

تطبيق على الأشجار الثنائية: شيفرة هوفمان5



وبالمثل نتابع فنحصل على الشجرة التالية:

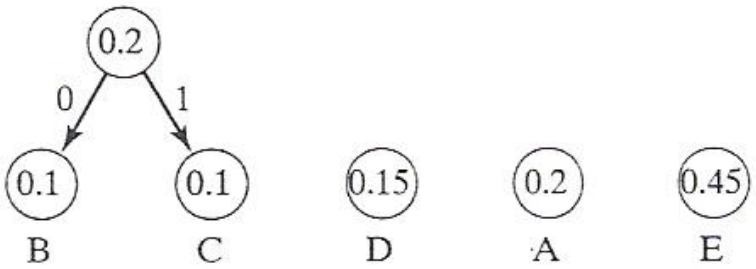


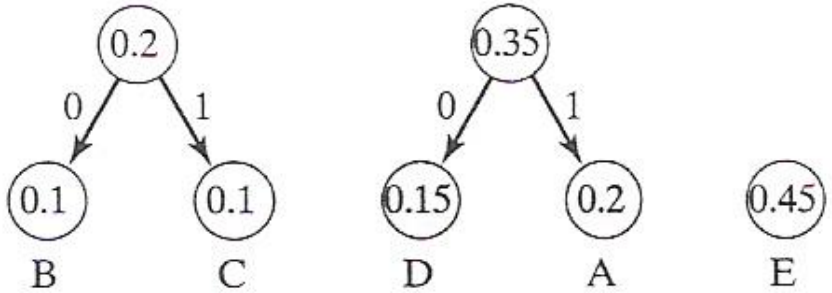
وبالتالي تكون شيفرة هوفمان الناتجة عن هذه الشجرة لهذه الأحرف كما يلي:

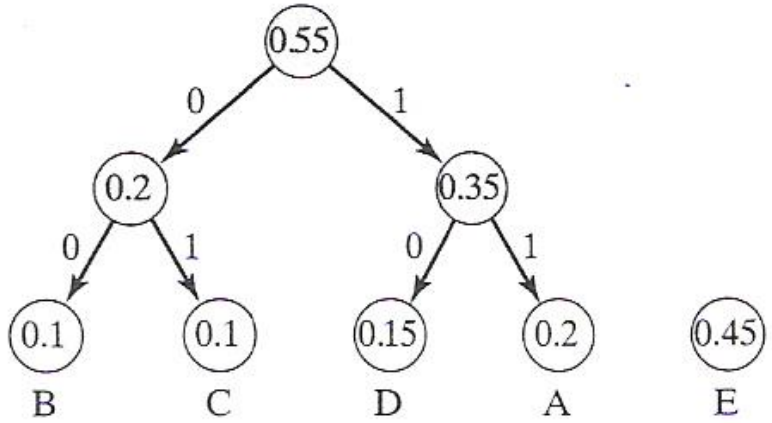
وكما بينا سابقاً فإن الطول المتوقع لشيفرة كل حرف هو2.1.

character	A	B	C	D	E
Huffman code	01	0000	0001	001	1

application of binary trees: Huffman codes

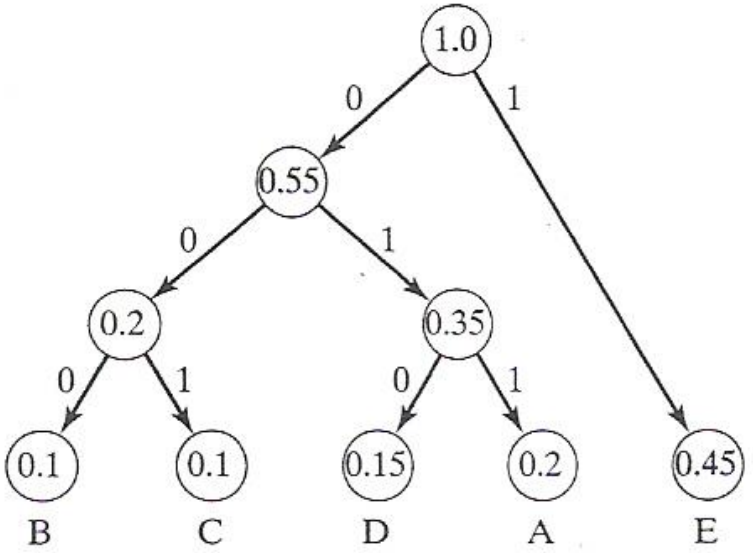






تطبيق على الأشجار الثنائية: شيفرة هوفمان6

يمكن تكوين شيفرة هوفمان بطريقة أخرى
نكتفي بعرضها من خلال الرسوم كما يلي:



وتكون الشيفرة الناتجة كما يلي:

character	A	B	C	D	E
Huffman code	011	000	001	010	1

إن قابلية فك التشفير مباشرة واضحة في شيفرة هوفمان. كل محرف مرتبط بعقدة ورقة في شجرة هوفمان وهناك مسار وحيد من الجذر إلى كل ورقة. وبالتالي ليس هناك تتالي من البتات يتضمن شيفرة لمحرف يمكن أن يكون جزءاً سابقاً من تتالي أطول من البتات لمحرف آخر.

إن خوارزمية فك التشفير سهلة جداً بسبب ميزة قابلية فك التشفير مباشرة:

HUFFMAN DECODING ALGORITHM

1-initialize pointer p to the root of the Huffman tree.

2-while the end of the message string has not been reached , do the following:

a- let x be the next bit in the string.

b- if x=0 then set p equal to its left child pointer.

else set p equal to its right child pointer.

c-if p points to a leaf then

i-display the character associated with that leaf.

ii-reset p to the root of the Huffman tree.

تطبيق على الأشجار الثنائية: شيفرة هوفمان8

كتوضيح، لنفرض أن الرسالة التالية:

0	1	0	1	0	1	1	0	1	0

قد استلمت، وأن هذه الرسالة مشفرة باستخدام شجرة هوفمان الثانية المبنية سابقاً، يتبع المؤشر المسار التالي 010 من جذر هذه الشجرة يتم اكتشافه وجعله جذر الشجرة:

0	1	0	1	0	1	1	0	1	0
	D								

0	1	0	1	0	1	1	0	1	0
	D		E						

الخانة التالية 1 تقود مباشرة إلى المحرف E:

0	1	0	1	0	1	1	0	1	0
	D		E		A			D	

وهكذا إلى أن تصبح الرسالة من الشكل: