C++ Plus Data Structures

Nell Dale

David Teague

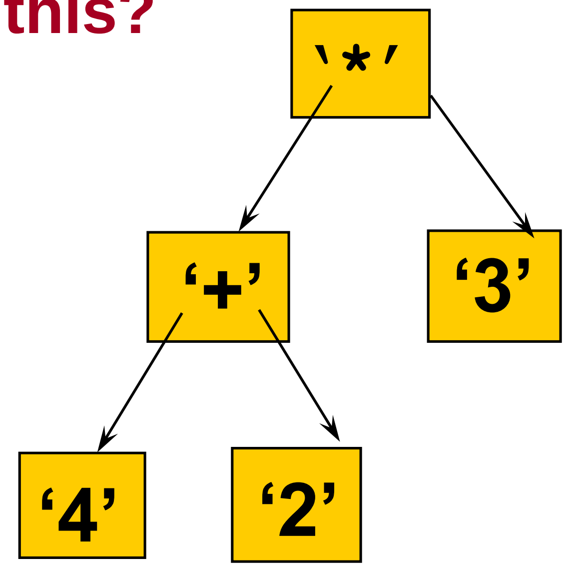
Chapter 9

Trees Plus

A Binary Expression Tree

How to evaluate this?

(4 + 2) * 3



What value does it have?

(4 + 2) * 3 = 18

A Binary Expression Tree is . . .

A special kind of binary tree in which:

1. Each leaf node contains a single operand,

2. Each nonleaf node contains a single binary operator, and

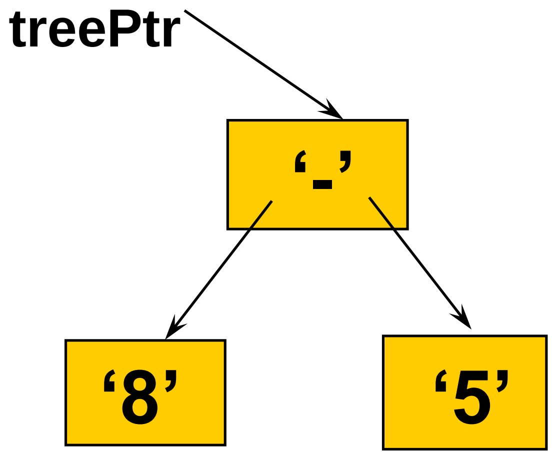
3. The left and right subtrees of an operator node represent subexpressions that must be evaluated before applying the operator at the root of the subtree.

Levels Indicate Precedence

When a binary expression tree is used to represent an expression, the levels of the nodes in the tree indicate their relative precedence of evaluation.

Operations at higher levels of the tree are evaluated later than those below them. The operation at the root is always the last operation performed.

A Two-Level Binary Expression

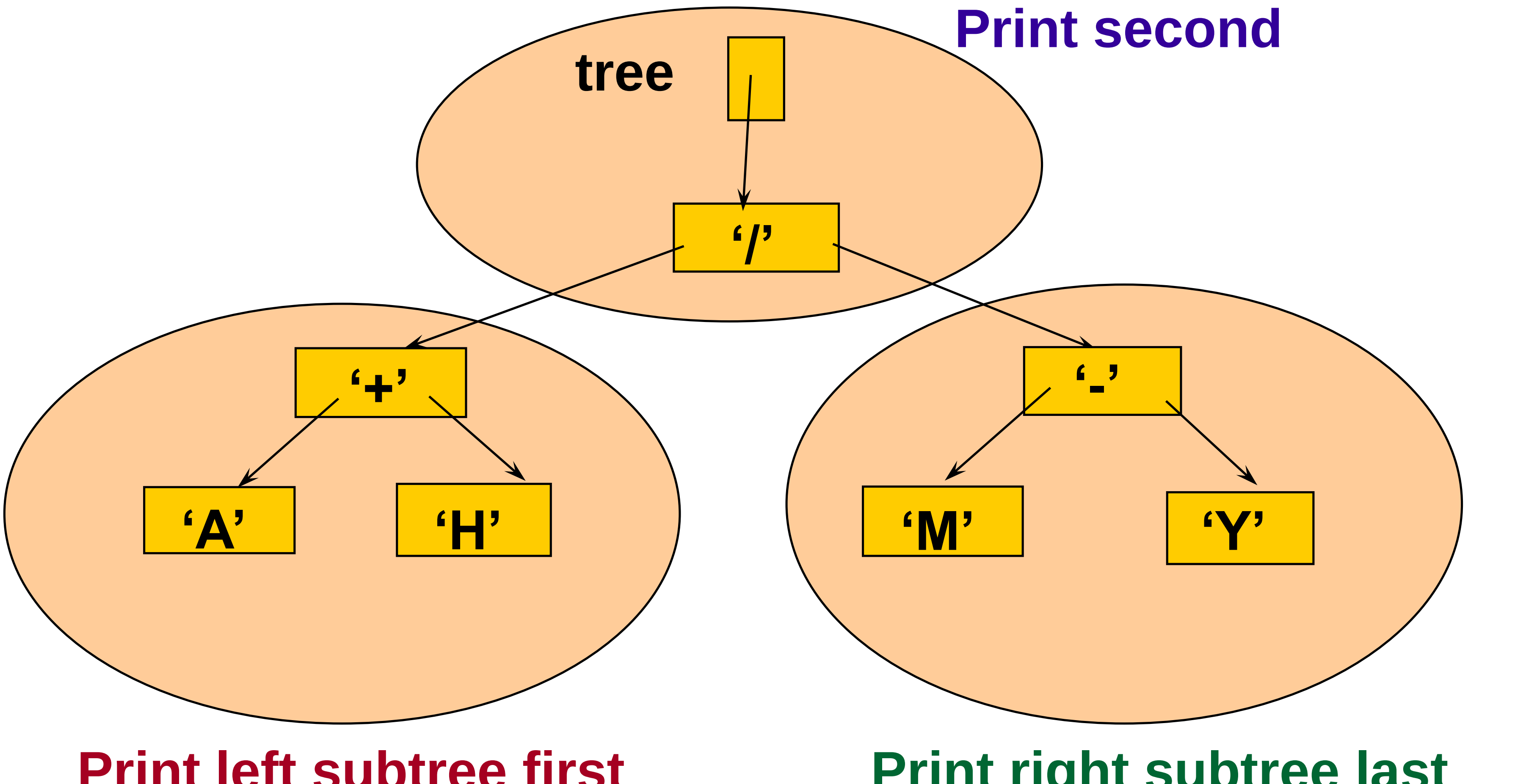


INORDER TRAVERSAL: `8 - 5 has value 3`

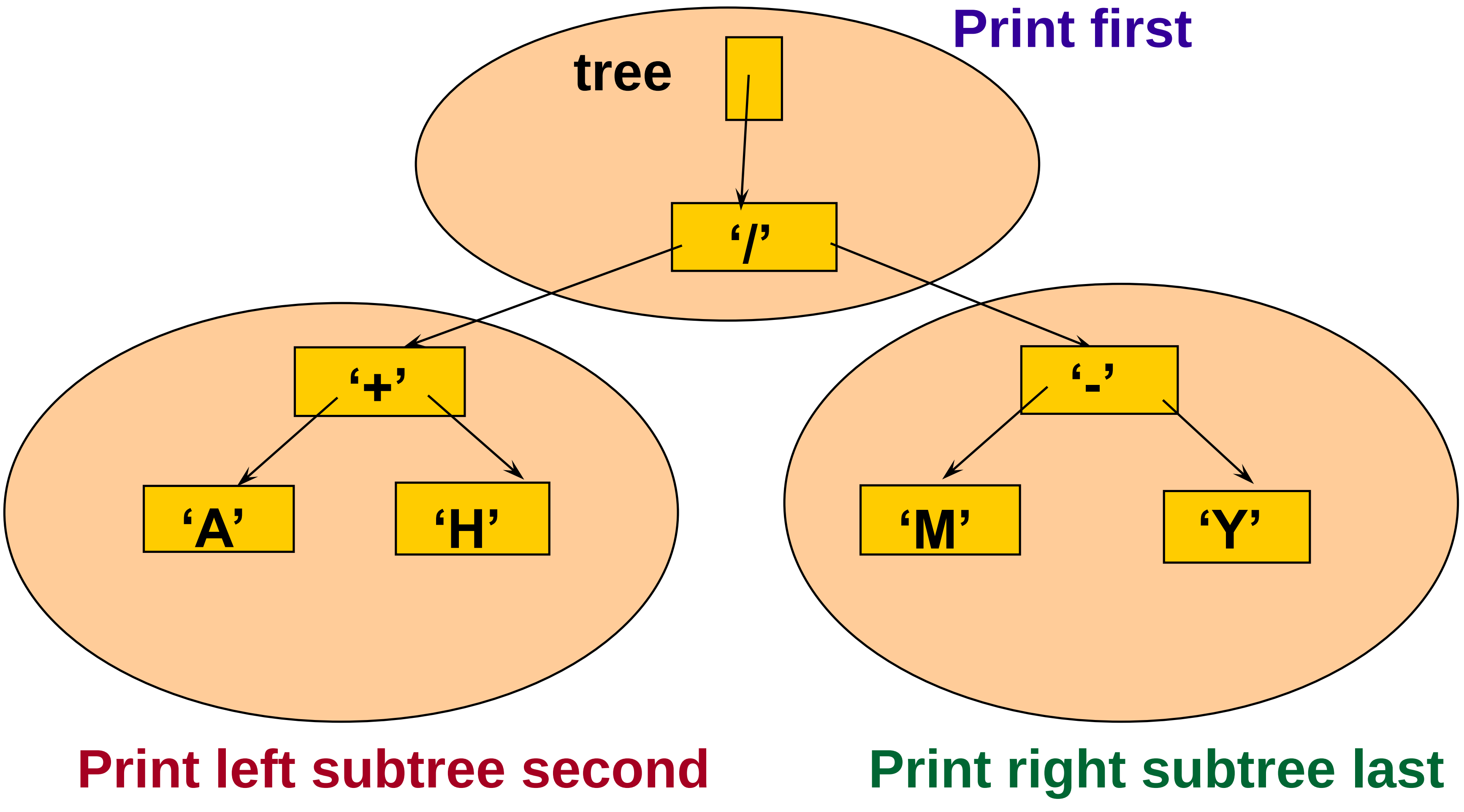
PREORDER TRAVERSAL: `- 8 5`

POSTORDER TRAVERSAL: `8 5 -`

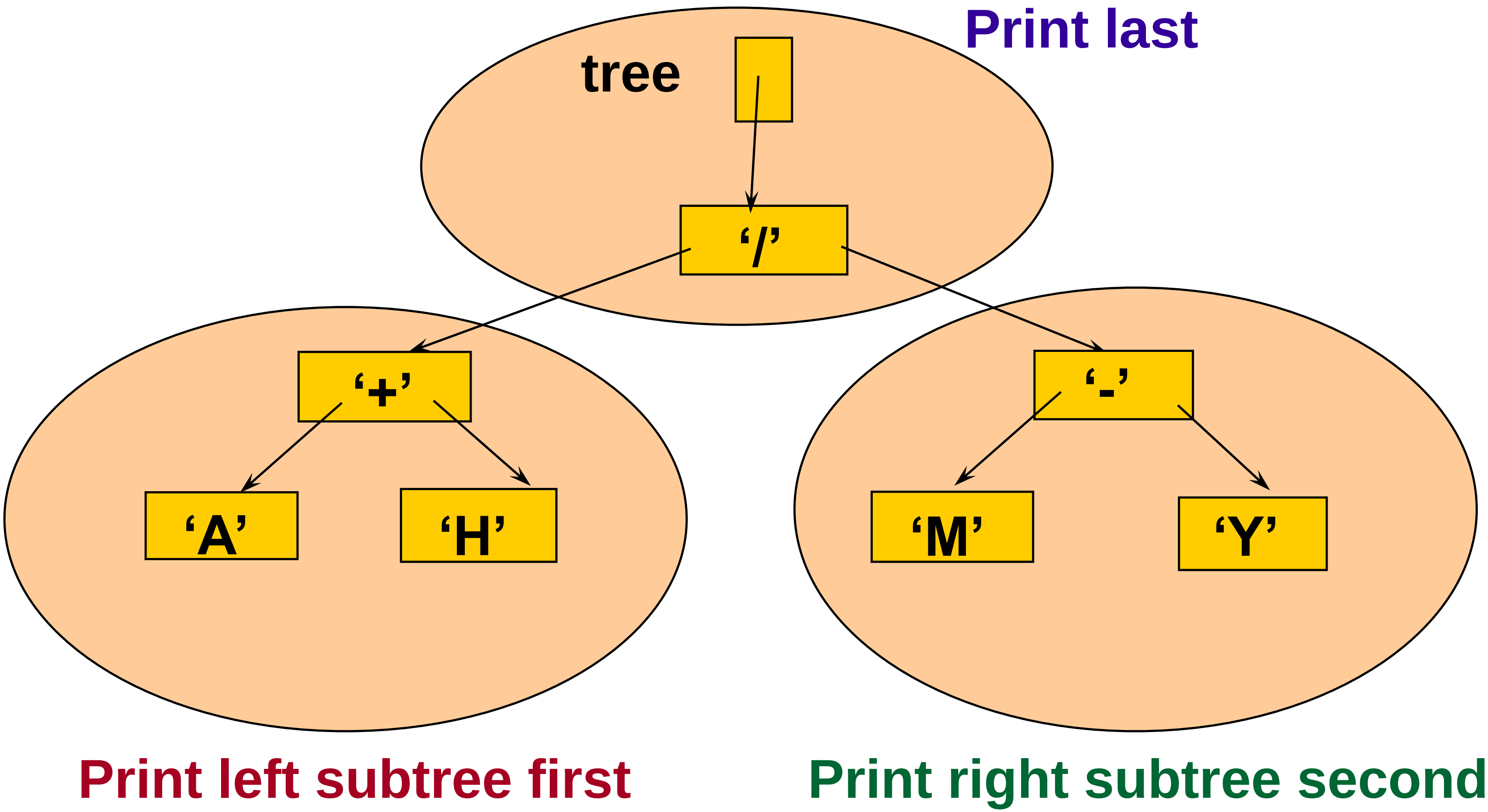
Inorder Traversal: (A + H) / (M - Y)



Preorder Traversal: / + A H - M Y

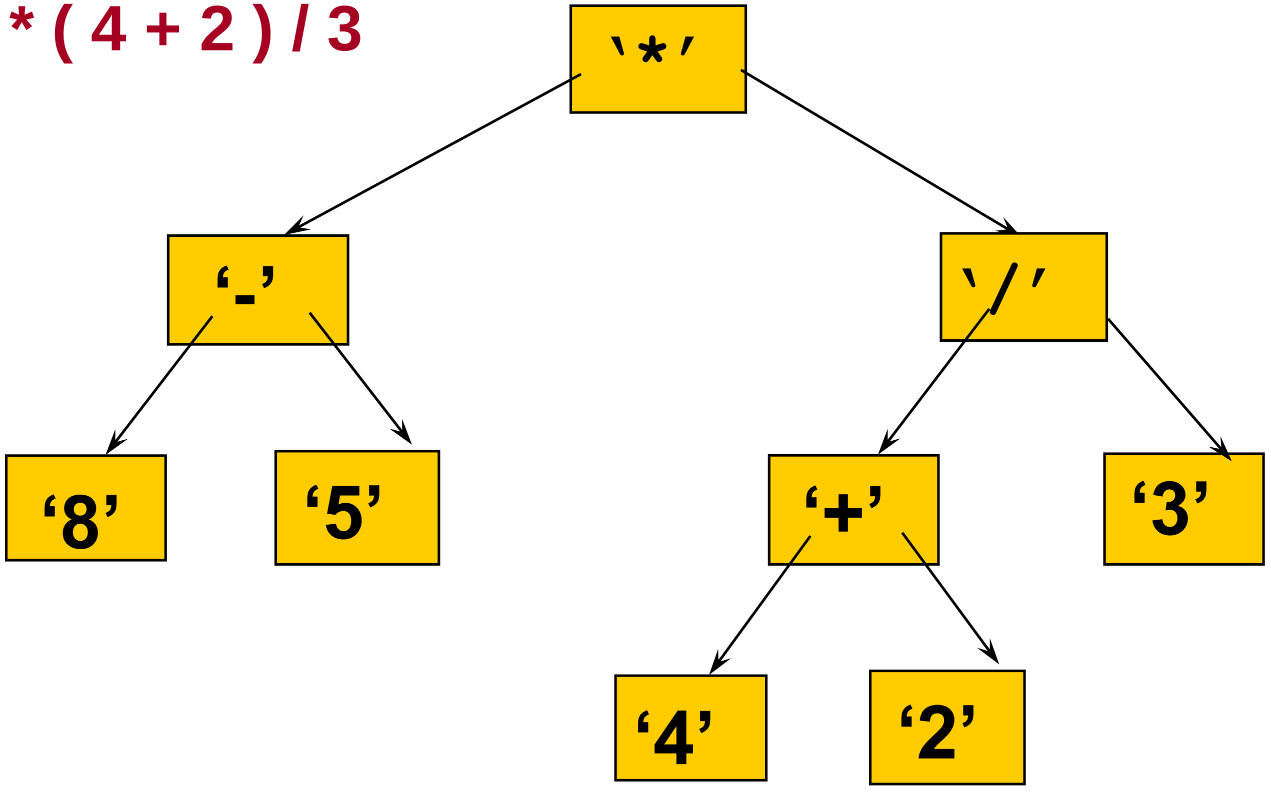


Postorder Traversal: A H + M Y - /



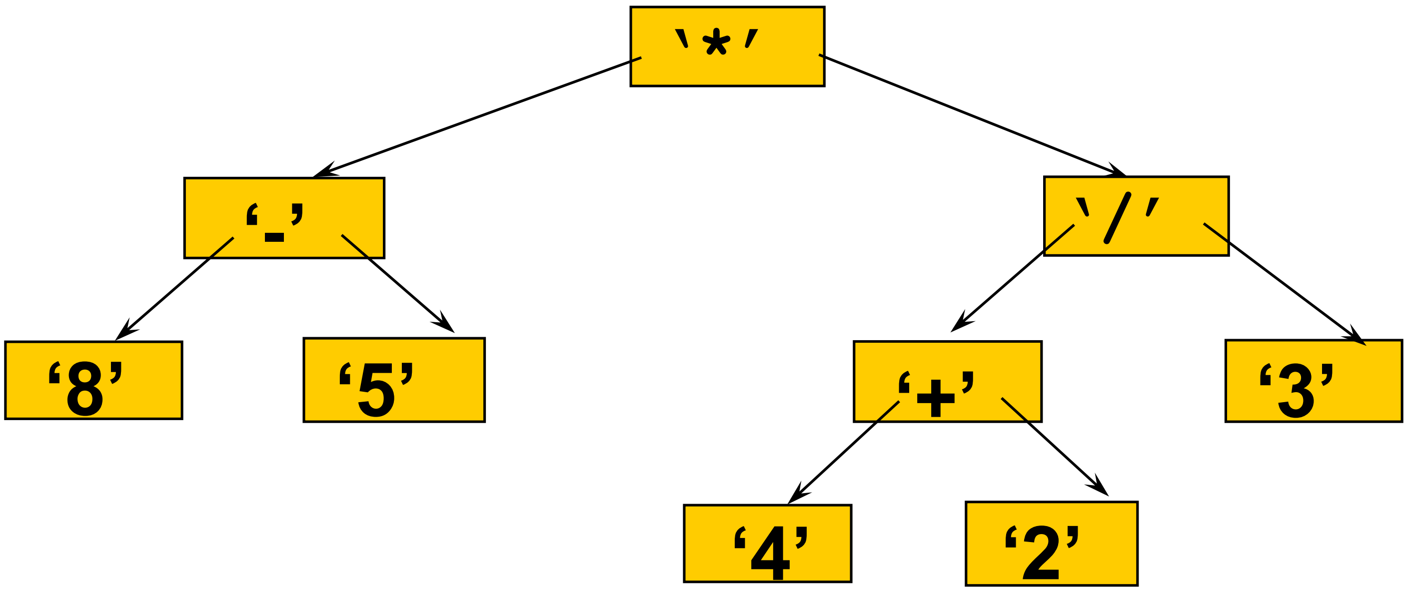
Evaluate
this binary expression tree

(8 - 5) * (4 + 2) / 3



What infix, prefix, postfix expressions does it represent?

A binary expression tree



Infix: ((8 - 5) * ((4 + 2) / 3))

Prefix: * - 8 5 / + 4 2 3

Postfix: 8 5 - 4 2 + 3 / * ***has operators in order used***

InfoNode has 2 forms

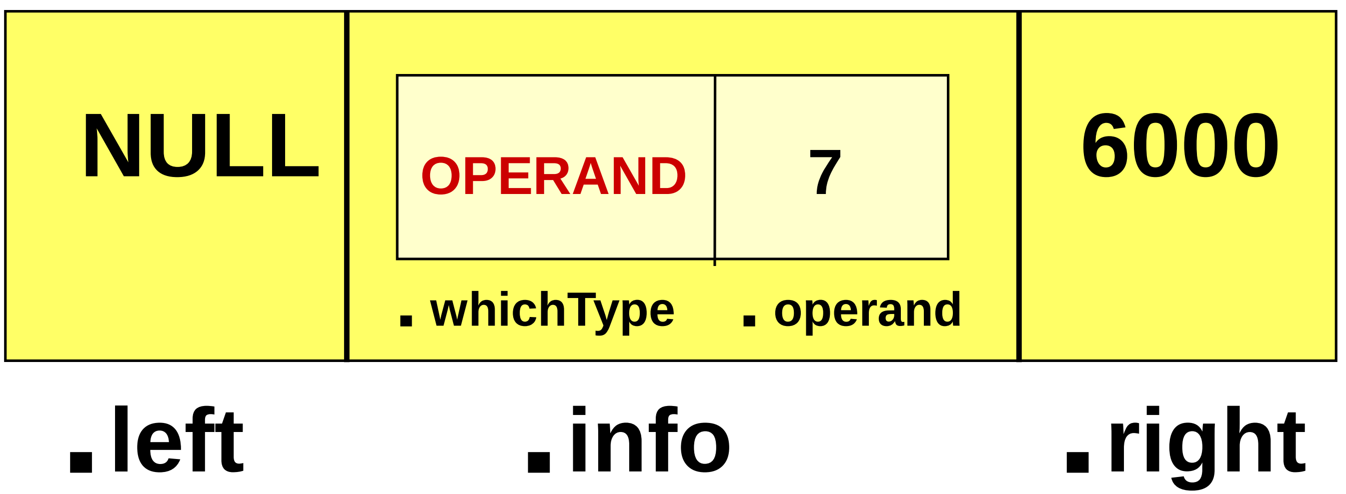
```
enum  OpType  { OPERATOR,  OPERAND } ;
struct   InfoNode
{
    OpType       whichType;
    union                                      // ANONYMOUS union
    {
          char    operation ;
          int      operand ;
    }
};
```

OPERATOR	'+'
. whichType	. operation

OPERAND	7
. whichType	. operand

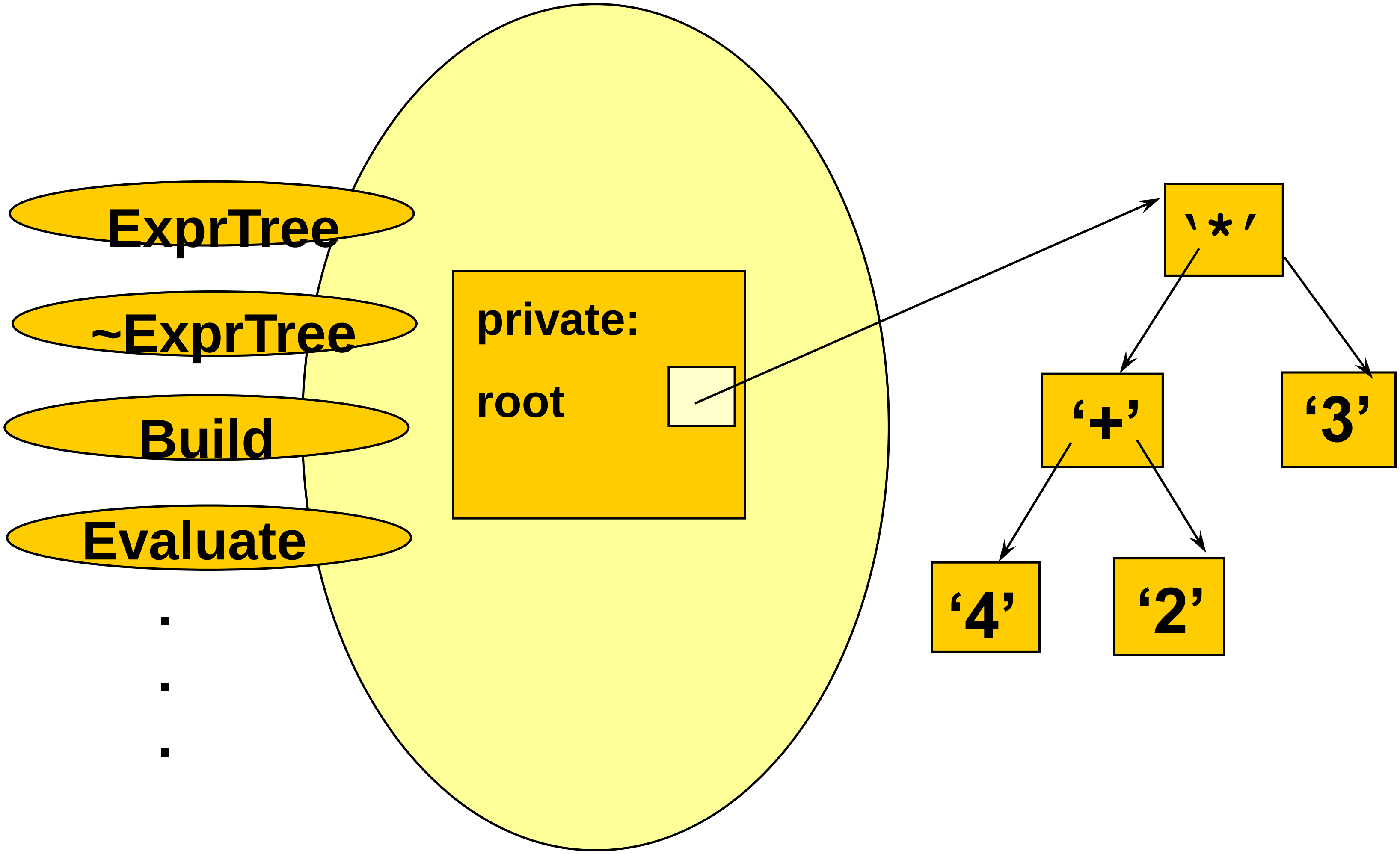
Each node contains two pointers

```
struct   TreeNode
{
   InfoNode     info ;                  // Data member
   TreeNode*   left ;                   // Pointer to left child
   TreeNode*   right ;                  // Pointer to right child
};
```



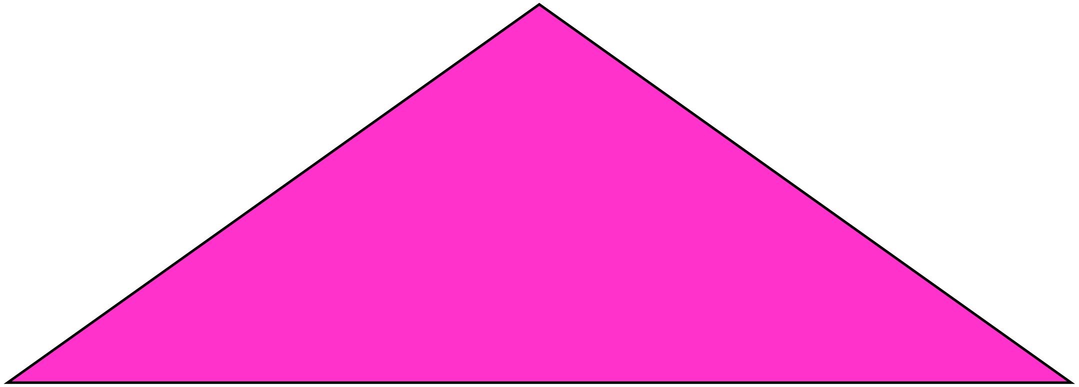
```
int   Eval ( TreeNode*  ptr )

// Pre:     ptr is a pointer to a binary expression tree.
// Post:   Function value = the value of the expression represented
//            by the binary tree pointed to by ptr.

{     switch  ( ptr->info.whichType )
      {
            case OPERAND :  return  ptr->info.operand ;
            case OPERATOR :
               switch ( tree->info.operation )
               {
                  case  '+'  :  return  ( Eval ( ptr->left ) + Eval ( ptr->right ) ) ;

                  case  '-'  :  return  ( Eval ( ptr->left )  -  Eval ( ptr->right ) ) ;

                  case  '*'  :  return  ( Eval ( ptr->left )  *  Eval ( ptr->right ) ) ;

                  case  '/'  :  return  ( Eval ( ptr->left )  /  Eval ( ptr->right ) ) ;
               }
      }
}
```

class ExprTree



A full binary tree

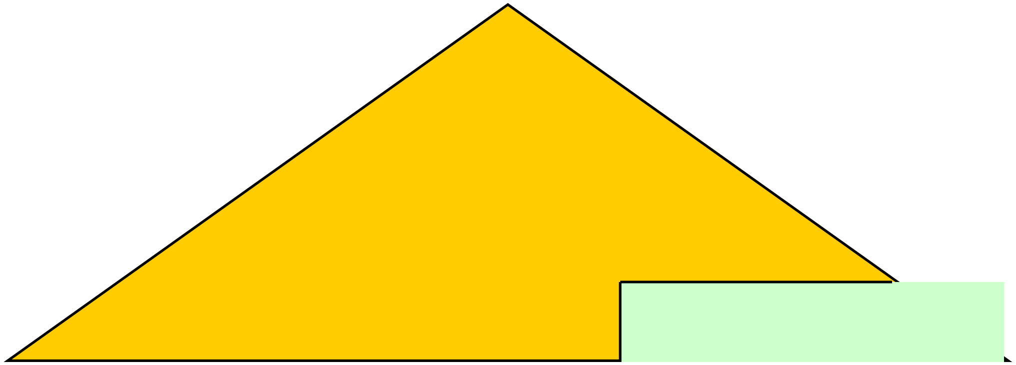
A **full binary tree** is a binary tree in which all the leaves are on the same level and every non leaf node has two children.



SHAPE OF A FULL BINARY TREE

A complete binary tree

A **complete binary tree** is a binary tree that is either full or full through the next-to-last level, with the leaves on the last level as far to the left as possible.



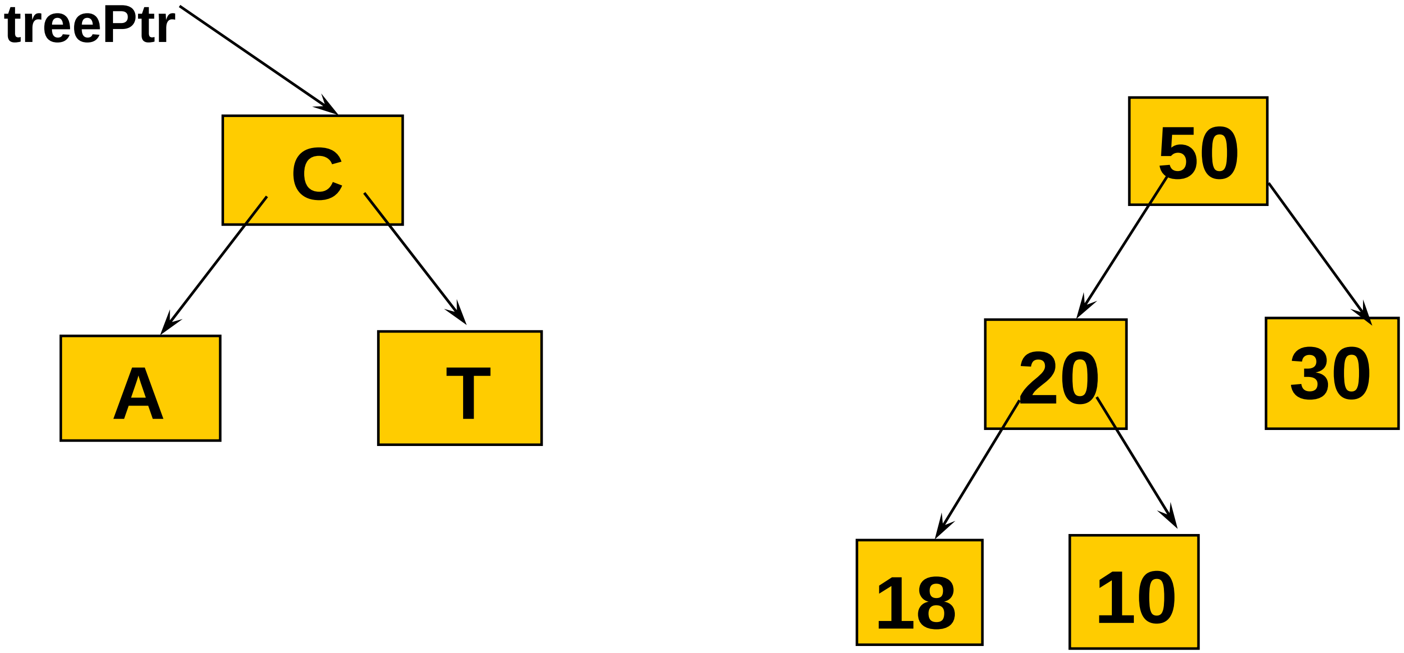
SHAPE OF A COMPLETE BINARY TREE

What is a Heap?

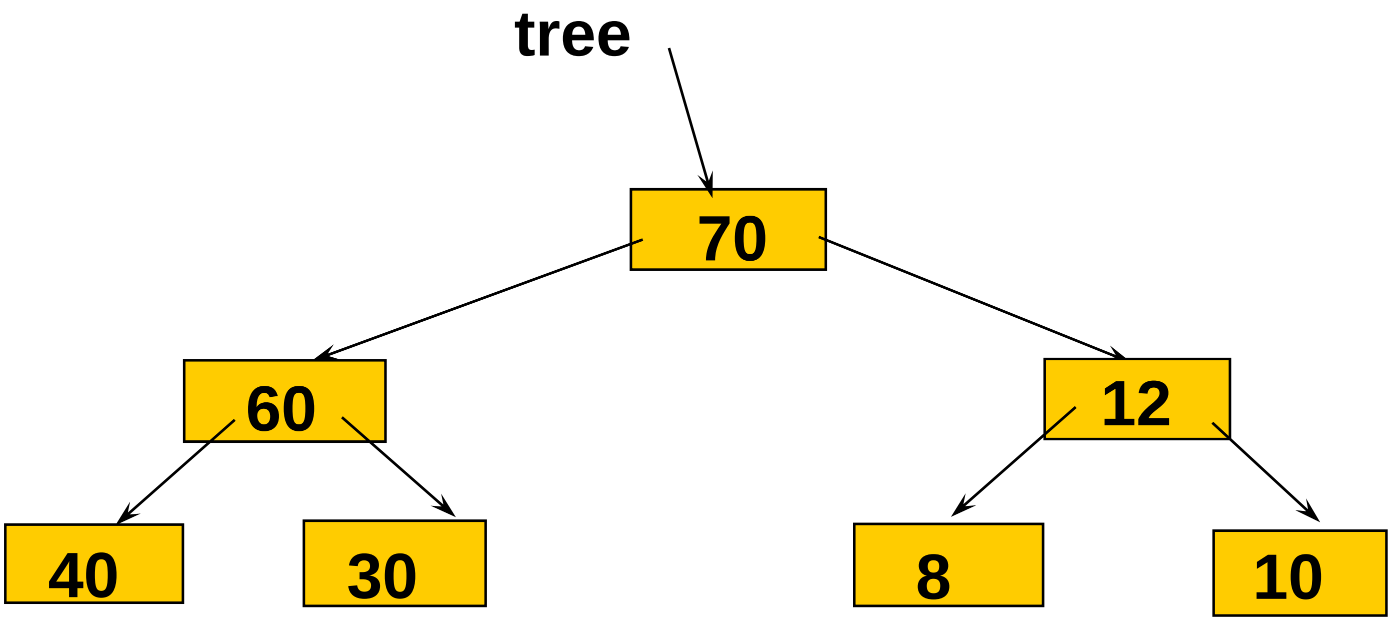
A heap is a binary tree that satisfies these special SHAPE and ORDER properties:

- **Its shape must be a complete binary tree.**
- **For each node in the heap, the value stored in that node is greater than or equal to the value in each of its children.**

Are these both heaps?



Is this a heap?

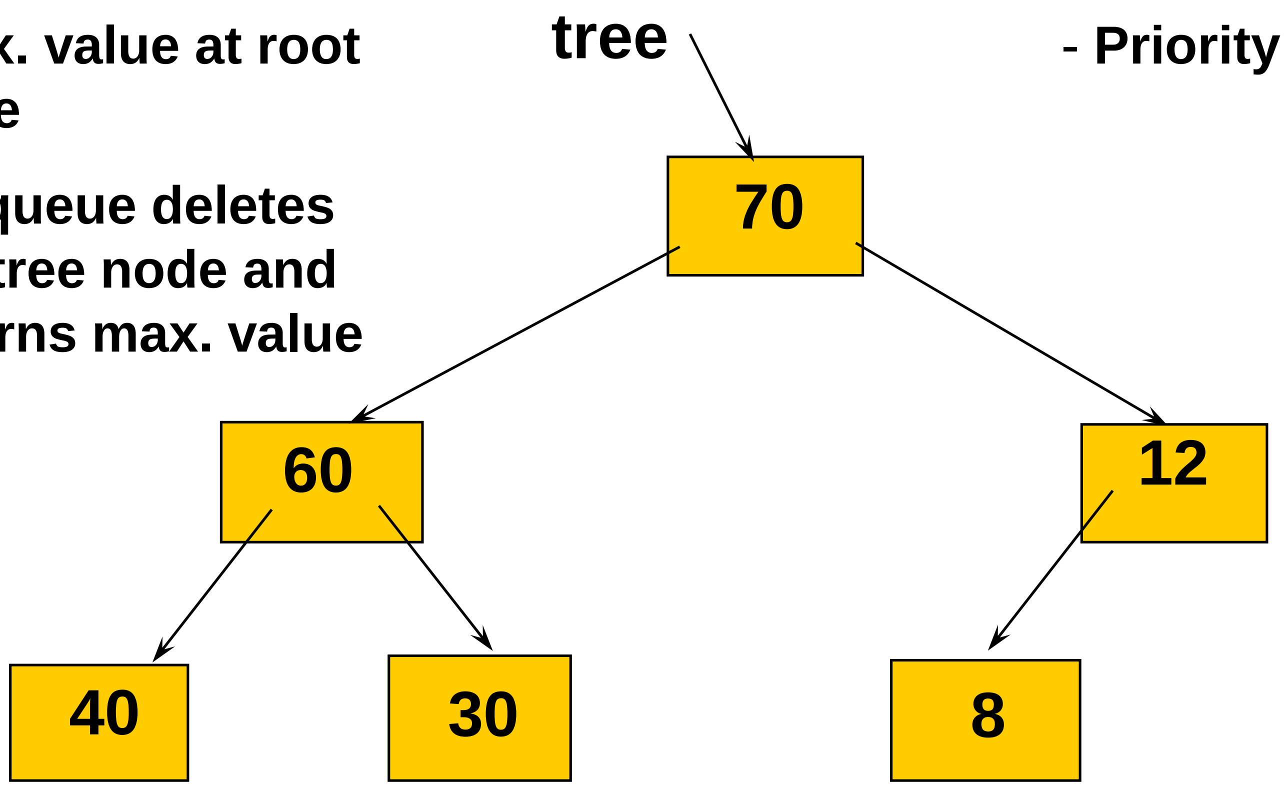


Where is the largest element in a heap always found?

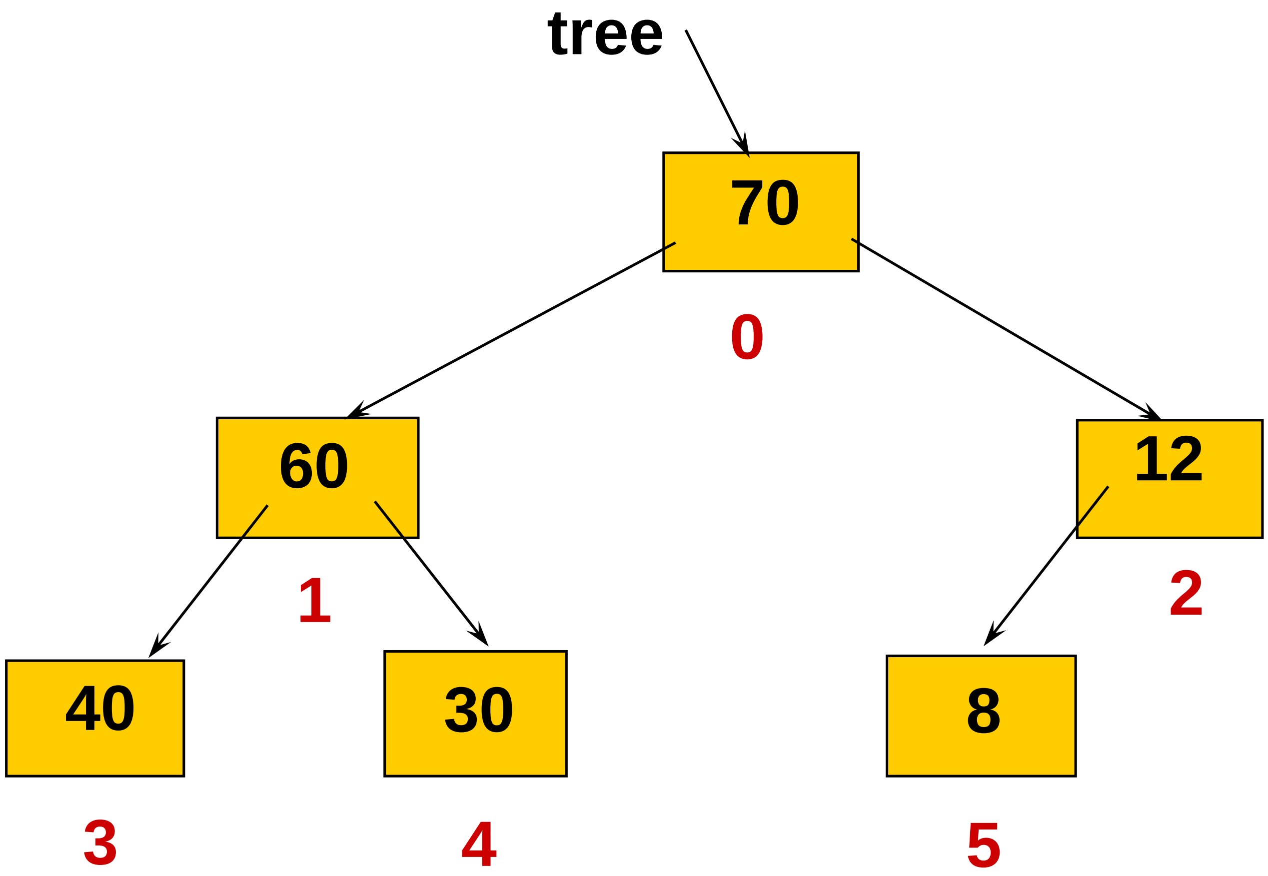
-Max. value at root node

-Dequeue deletes the tree node and returns max. value

- Priority Queue



We can number the nodes left to right by level this way



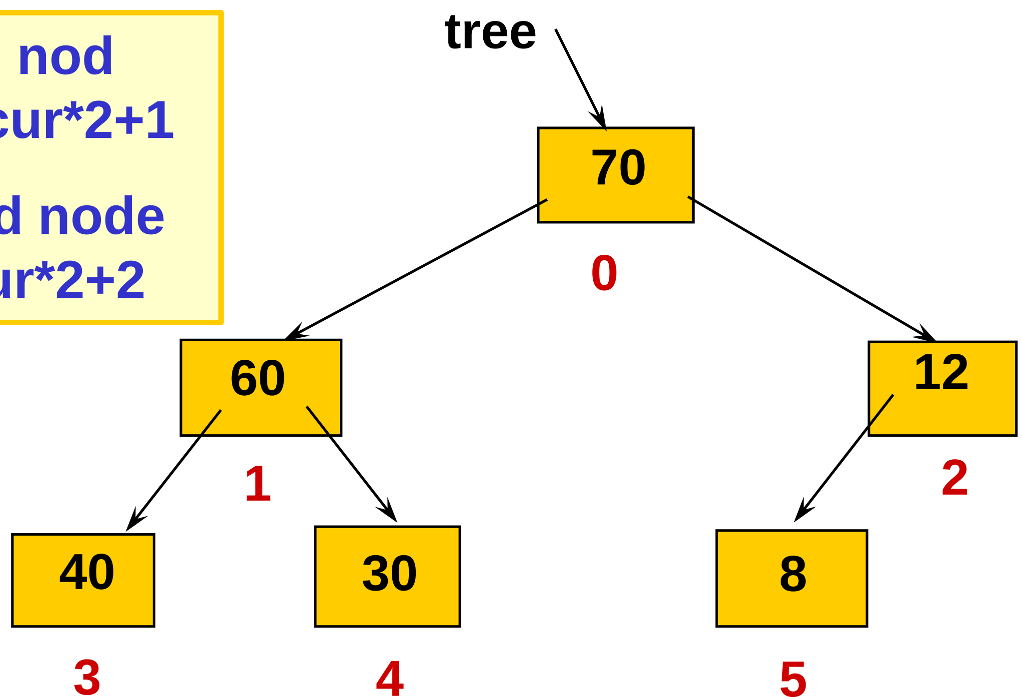
And use the numbers as array indexes to store the tree

tree.nodes

[0]	70
[1]	60
[2]	12
[3]	40
[4]	30
[5]	8
[6]	

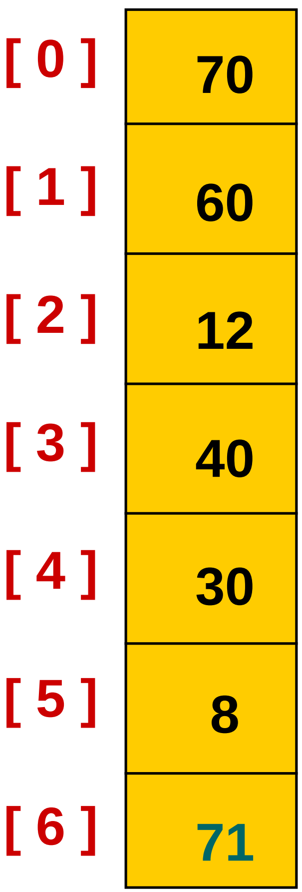
Left child nod
Index = Icur*2+1

Righ child node
Index=Icur*2+2

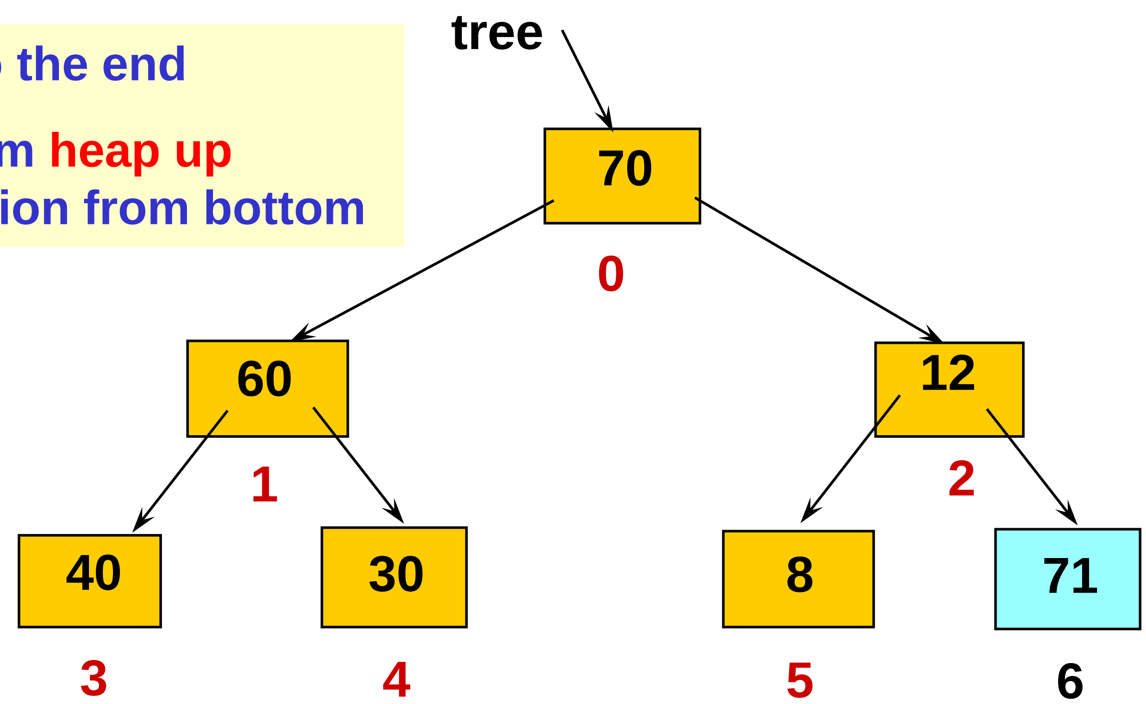


Add New Element "71"

tree.nodes



- **Add to the end**
- **Perform heap up operation from bottom**

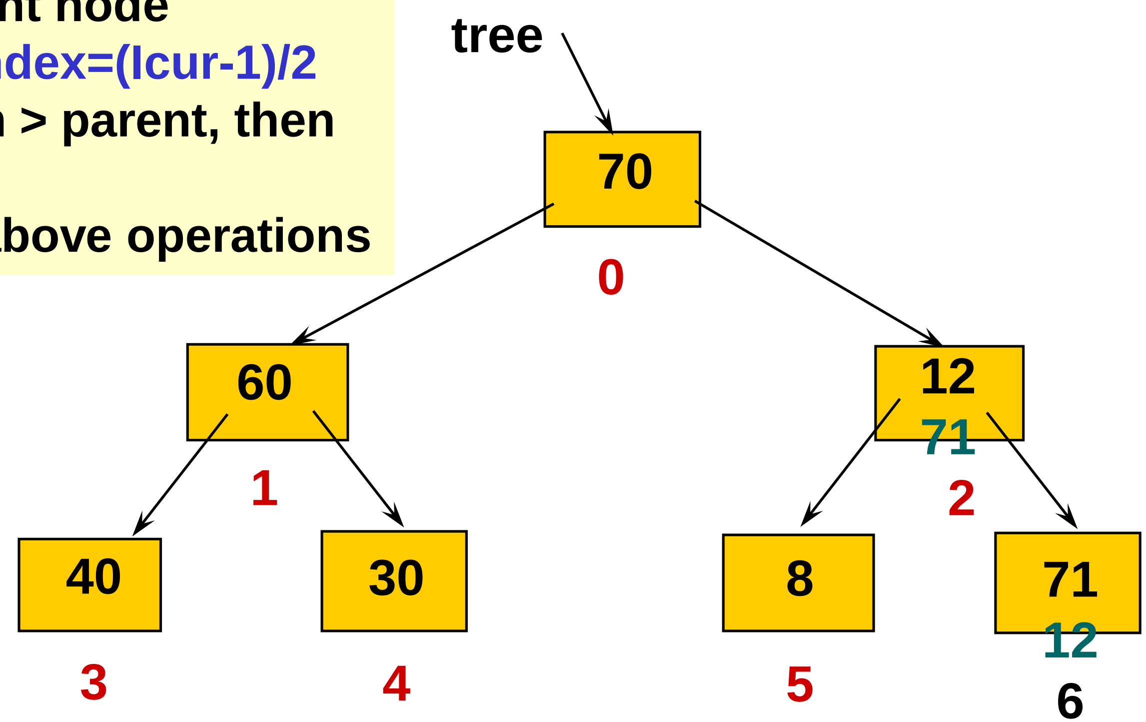


Add New Element “71”

tree.nodes

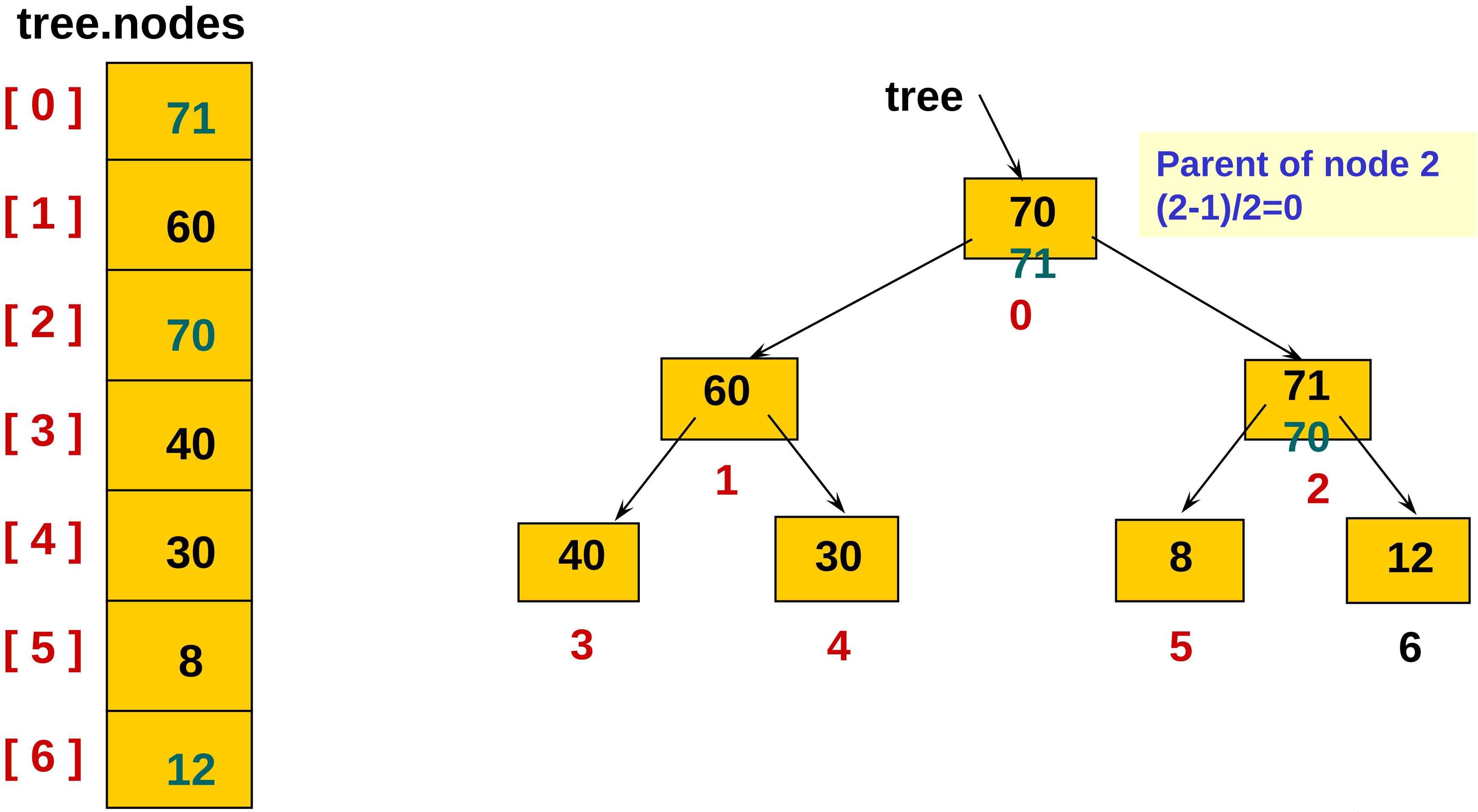
[0]	70
[1]	60
[2]	71
[3]	40
[4]	30
[5]	8
[6]	12

- Get parent node
  parent Index=(Icur-1)/2
- If bottom > parent, then
  swap
- Repeat above operations



Parent of node 6
(6-1)/2=2

Add New Element “71”

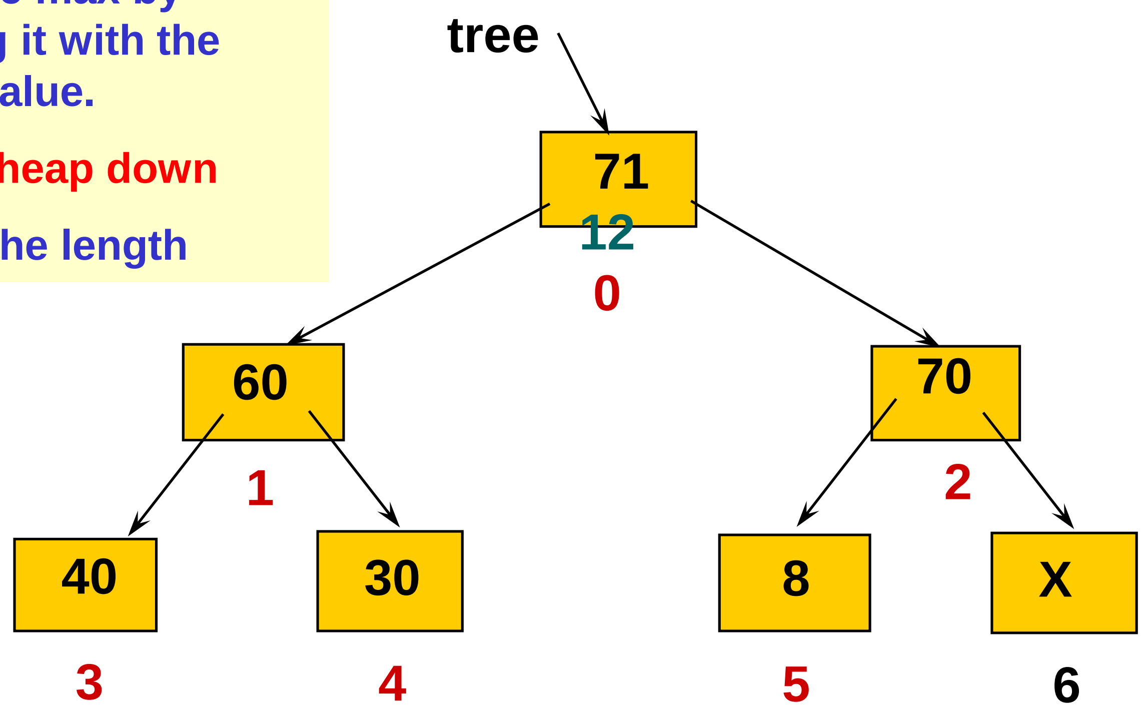


Delete Max. Element “71”

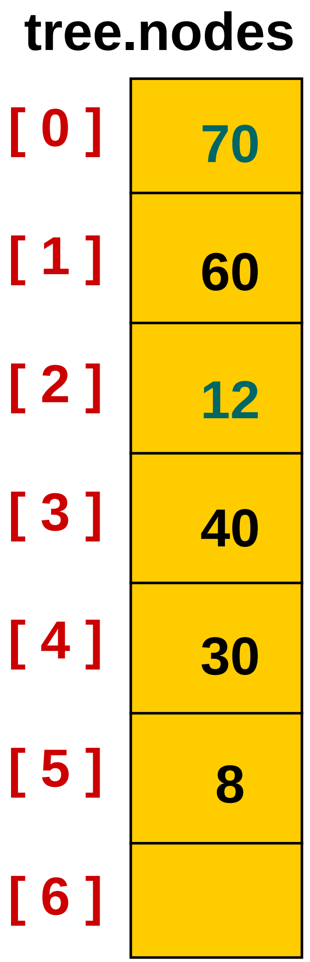
tree.nodes

[0]	12
[1]	60
[2]	70
[3]	40
[4]	30
[5]	8
[6]	

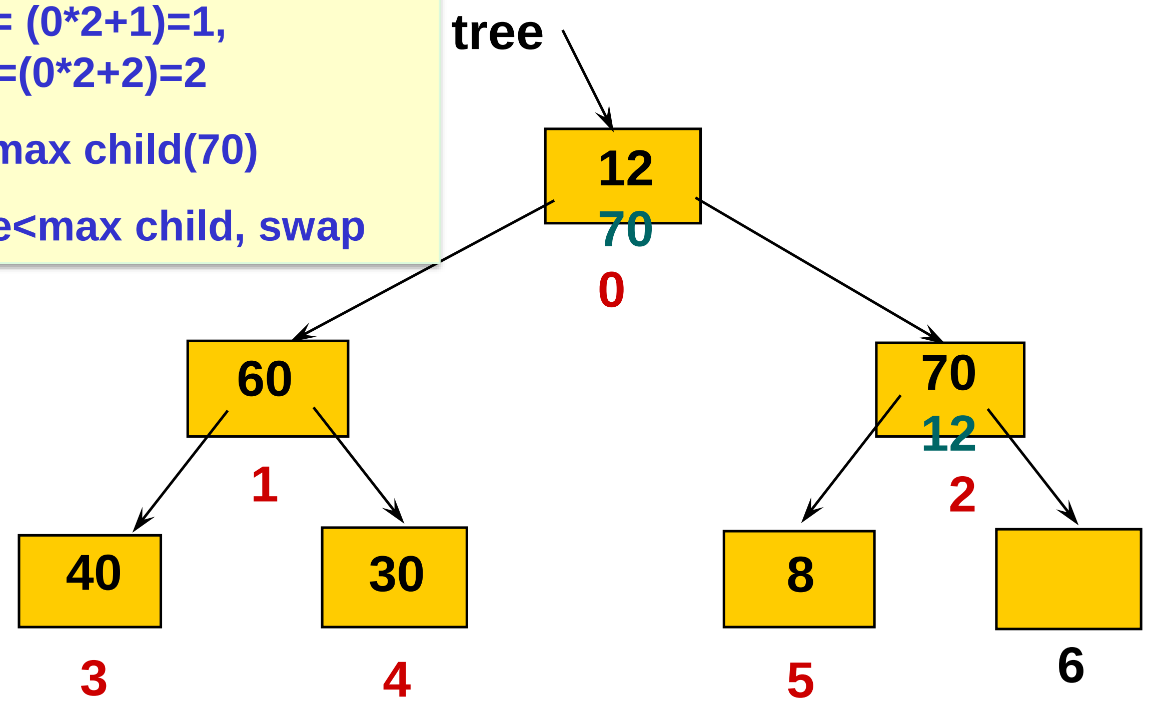
- Delete the max by replacing it with the bottom value.
- Perform heap down
- Reduce the length



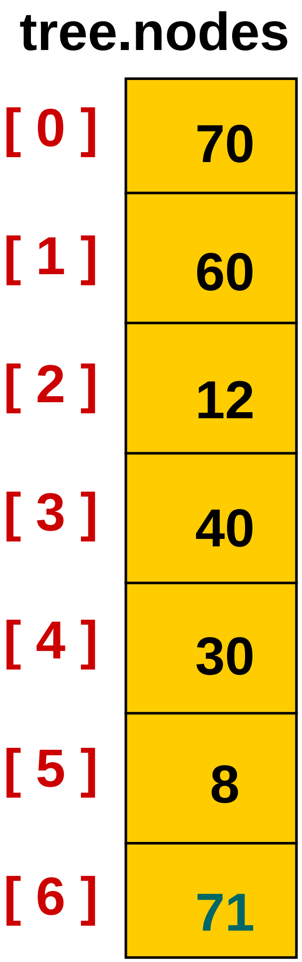
Delete Max. Element



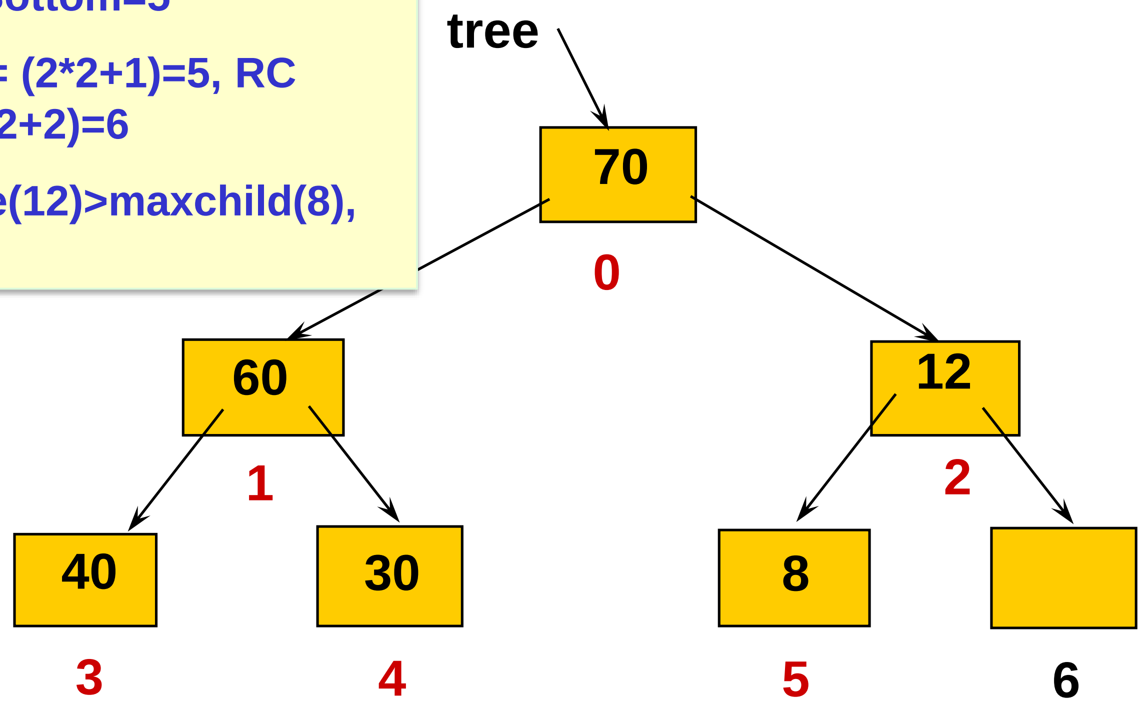
- Root=0, Bottom=5
- LC index= (0*2+1)=1,
  RC index=(0*2+2)=2
- Find the max child(70)
- Since tree<max child, swap



Delete Max. Element



- Root=2, Bottom=5
- LC index= (2*2+1)=5, RC index=(2*2+2)=6
- Since tree(12)>maxchild(8), end



```
//  HEAP SPECIFICATION

//  Assumes  ItemType  is either a built-in simple data type
//  or a class with overloaded rational operators.

template< class  ItemType >
struct   HeapType
{
   void   ReheapDown ( int  root ,  int  bottom ) ;
   void   ReheapUp ( int  root,  int  bottom ) ;

   ItemType*  elements ;     // ARRAY to be allocated dynamically
   int  numElements ;
};
```

ReheapDown

```
// IMPLEMENTATION  OF RECURSIVE HEAP MEMBER FUNCTIONS
template< class  ItemType >
void   HeapType<ItemType>::ReheapDown ( int  CurRoot,  int
bottom )

//  Pre:  root is the index of the node that may violate the heap
//          order property
//  Post:  Heap order property is restored between root and bottom

{
    int  maxChild ;
    int  rightChild ;
    int  leftChild ;

    leftChild  =  CurRoot * 2 + 1 ;
    rightChild  =  CurRoot * 2 + 2 ;
```

```
    if  ( leftChild  <=  bottom )             // ReheapDown continued
    {
        if  ( leftChild  == bottom )
            maxChild  =  leftChld ;
        else
        {
            if  (elements [ leftChild ] <= elements [ rightChild ] )
                maxChild  =  rightChild ;
            else
                maxChild  =  leftChild ;
        }
        if  ( elements [ CurRoot ] < elements [ maxChild ] )
        {
            Swap ( elements [ CurRoot ] , elements [ maxChild ] ) ;
            ReheapDown ( maxChild, bottom ) ;
        }
    }
}
```

```
// IMPLEMENTATION              continued

template< class ItemType >
void   HeapType<ItemType>::ReheapUp ( int  root,  int  CurBottom )

// Pre:  bottom is the index of the node that may violate the heap
//        order property.  The order property is satisfied from root to
//         next-to-last node.
// Post:  Heap order property is restored between root and bottom

{
     int  parent ;

     if  ( CurBottom  > root )
     {
          parent = ( CurBottom - 1 ) / 2;
          if ( elements [ parent ]  <  elements [ CurBottom ] )
          {
                    Swap ( elements [ parent ], elements [ CurBottom ] ) ;
                    ReheapUp ( root, parent ) ;
          }
     }
}
```

Priority Queue

A priority queue is an ADT with the property that only the highest-priority element can be accessed at any time.

ADT Priority Queue Operations

Transformers

- MakeEmpty
- Enqueue
- Dequeue

change state

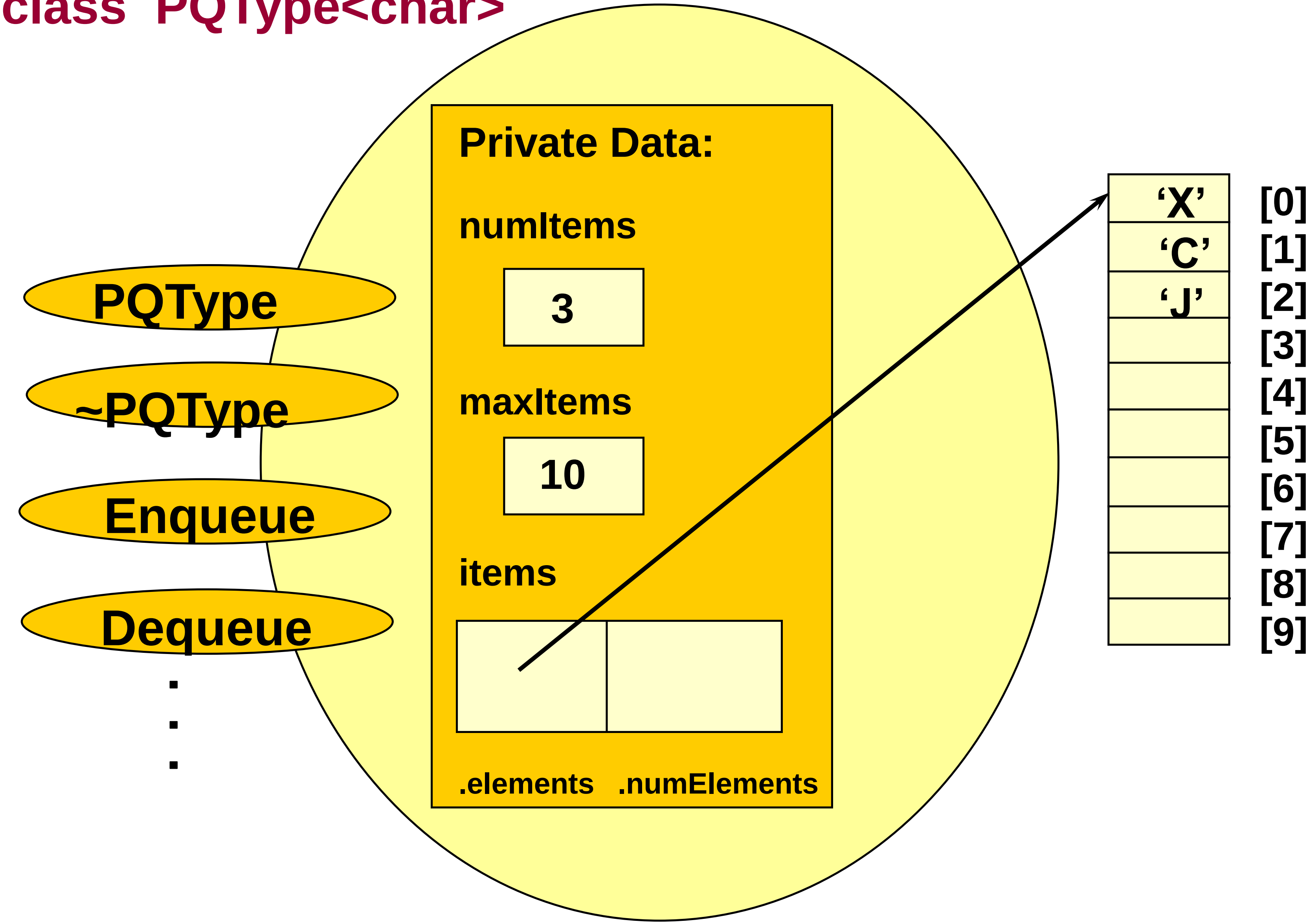
Observers

- IsEmpty
- IsFull

observe state

```
// CLASS PQTYPE DEFINITION AND MEMBER FUNCTIONS
//--------------------------------------------------------
#include "bool.h"
#include "ItemType.h"      // for ItemType

template<class ItemType>
class PQType  {
public:
   PQType( int );
   ~PQType ( );
   void MakeEmpty( );
   bool IsEmpty( ) const;
   bool IsFull( ) const;
   void Enqueue( ItemType item );
   void Dequeue( ItemType&  item );

private:
   int        length;
   HeapType<ItemType>  items;
   int        maxItems;
};
```

class PQType<char>



```
template<class ItemType>
Void PQType<ItemType>:: Dequeue(Itemtype& item)
// Post: element with highest priority has been
//removed from the queue; a copy is returned in item
{
  if (length==0)
      throw EmptyPQ();
  else
  {
      item = items.elements[0];
      items.elements[0] = items.elements[length-1];
      length--;
      items.ReheapDown(0,length-1);
  }
}
```

```
// Enqueue
#include "ItemType.h"        // for ItemType

template<class ItemType>
Void PQType<ItemType>:: Equeue(Itemtype newItem)
// Post: newItem is in the queueu
{
  if (length==maxItems)
      throw FullPQ();
  else
  {
      length++;
      items.elements[length-1] = newItem;
      items.ReheapUp(0,length-1);
  }
}
```

Comparison of Priority Queue

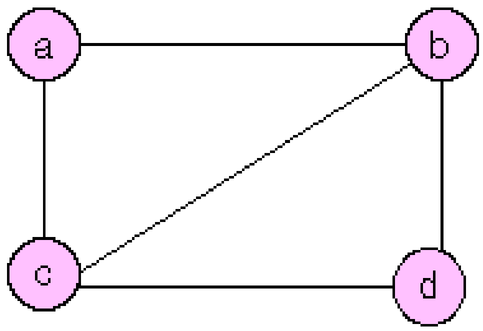
	Enqueue	Dequque
Heap	$O(\log_2 N)$	$O(\log_2 N)$
Linked List	$O(N)$	$O(1)$
Binary Search		
Balanced	$O(\log_2 N)$	$O(\log_2 N)$
Skewed	$O(N)$	$O(N)$

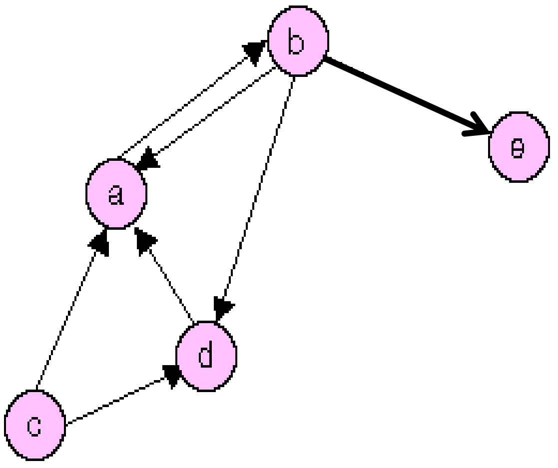
Graph

- Graph: A data structure that consists of a set of nodes and a set of edges that relate the nodes to one another
  - G = [V,E],
    - V(G): a finite, nonempty set of vertices
    - E(G): a set of edges
- Vertex: A node in graph
- Edge or arc: A pair of vertices representing a connection between nodes in a graph
  - (1,2) → edge connecting vertex 1 and 2

Types of graphs

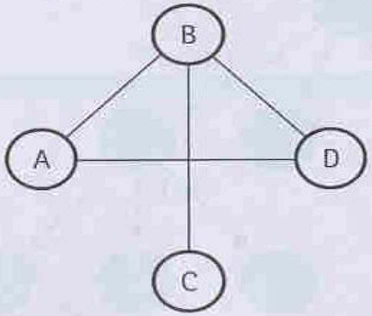
- Undirected Graph: A graph in which the edges have no direction
  - Edge (1,2) and edge (2,1) are same
- Directed Graph(Digraph): A graph in which each edge is directed from one vertex to another vertex.
  - Edge (1,2) starts from vertex 1 and points to vertex 2





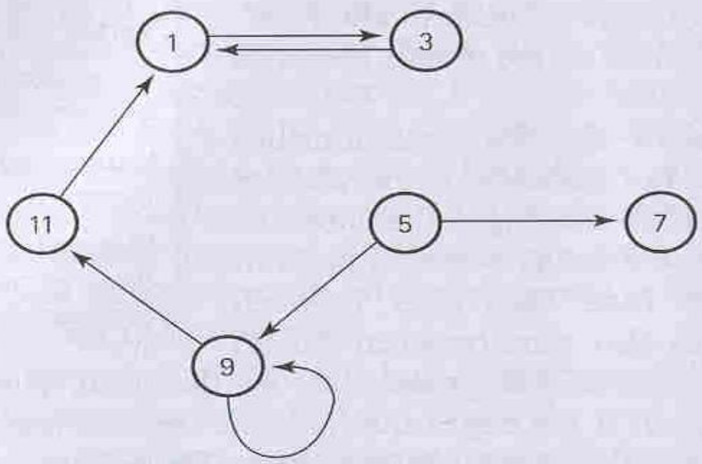
Some Examples

(a) Graph1 is an undirected graph.



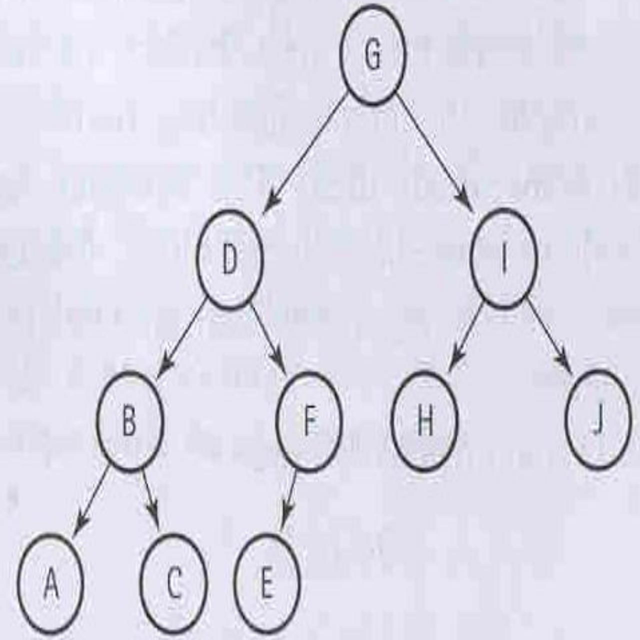
V(Graph1) = {A, B, C, D}
E(Graph1) = {(A, B), (A, D), (B, C), (B, D)}

(b) Graph2 is a directed graph.



V(Graph2) = {1, 3, 5, 7, 9, 11}
E(Graph2) = {(1, 3), (3, 1), (5, 7), (5, 9), (9, 11), (9, 9), (11, 1)}

(c) Graph3 is a directed graph.



V(Graph3) = {A, B, C, D, E, F, G, H, I, J}
E(Graph3) = {(G, D), (G, I), (D, B), (D, F), (I, H), (I, J), (B, A), (B, C), (F, E)}

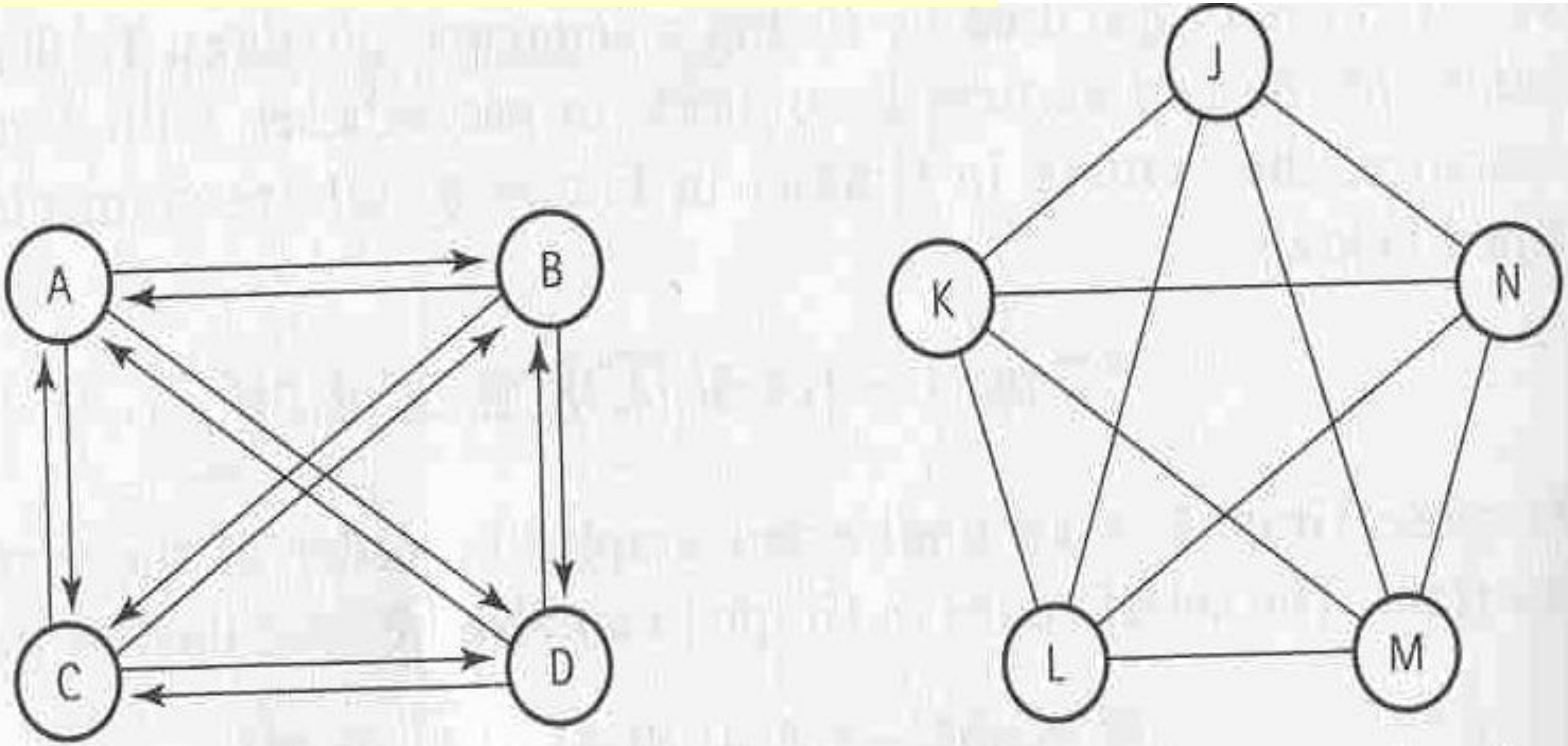
Figure 9.8 *Some examples of graphs*

Definition of terminology

- Adjacent vertices: Two vertices in a graph that are connected by an edge
- Path: A sequence of vertices that connects two nodes in a graph
- Complete graph: A graph in which every vertex is directly connected to every other vertex
- Weighted Graph: A graph in which each edge carries a value

Complete Graph

- Number of edges = n(n-1)/2



(a) Complete directed graph.

(b) Complete undirected graph.

Figure 9.9 *Two complete graphs*

A Weighted Graph

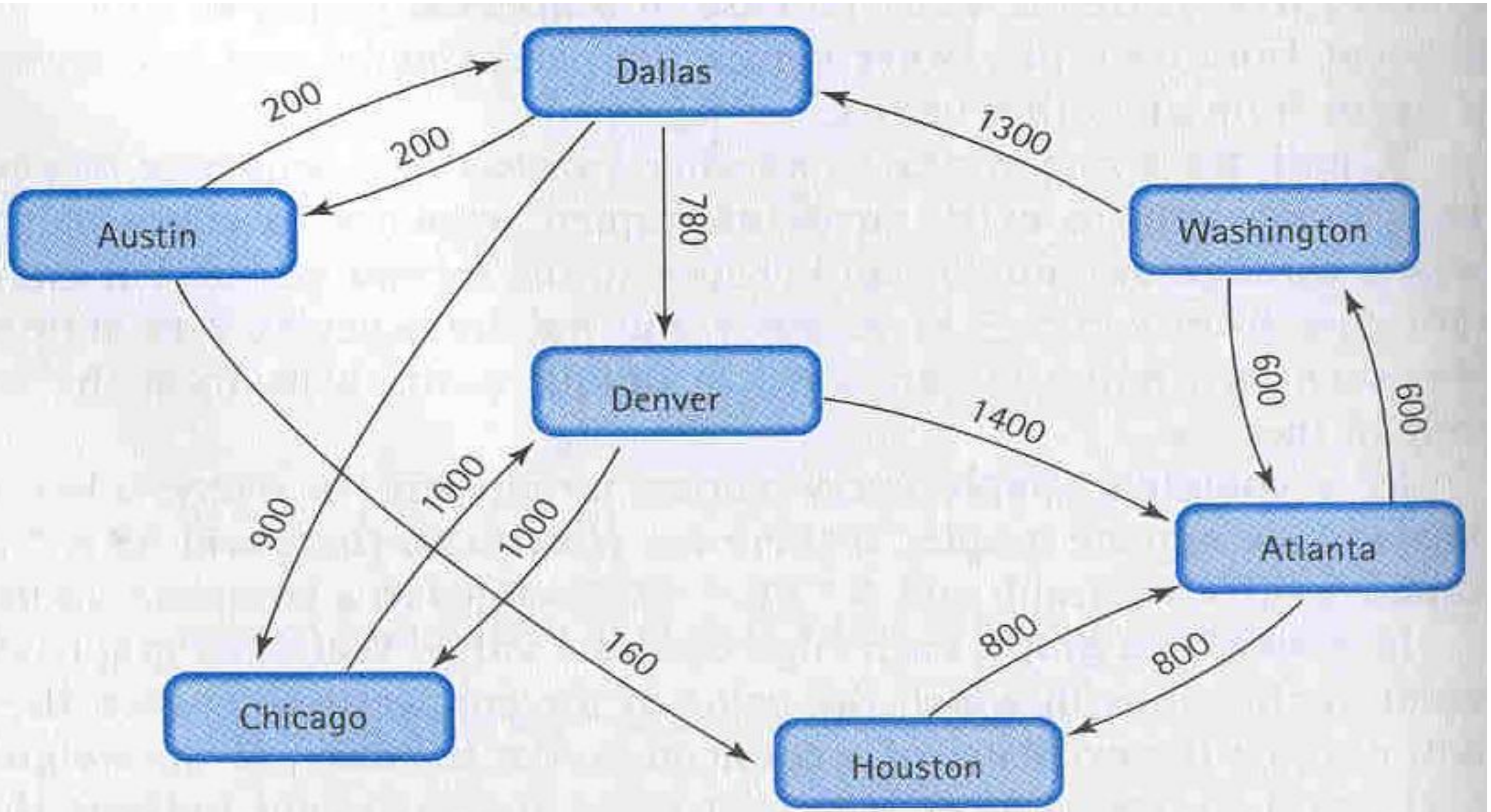


Figure 9.10 *A weighted graph*

Graph ADT

- MakeEmpty
- Boolean IsEmpty
- Boolean IsFull
- AddVertex(VertexType vertex)
- AddEdge(VertextType fromVertex, VertexType toVertex, EdgeValueType weight)
- EdgeValueType WeightIs(VetexType fromVertex, VertexType toVertex)
- GetToVertices(VertexType vertex, QueueType& vertexQ) // returns adjacent vertices in a queue

Depth First Search(깊이 우선 탐색)

```
Found = False
Stack.Push(startVertex)
Do
    stack.Pop(vertex)
    if vertex = endVertex
        write final vertex
        Found = True
    else
        Write this vertex
        Push all adjacent vertices onto stack(방문하지 않은)
While !stack.IsEmpty() AND !Found
If (!found)
    Write "Path does not exist"
```

Depth-First: Post order traversal과 같이 트리의 가장 밑**(**깊이**)**까지 내려간 다음 올라오면서 방문**.**

Breadth-First: 각 정점을 **level**별로 내려가면서 차례대로 방문

Depth-First: Austin to Washington

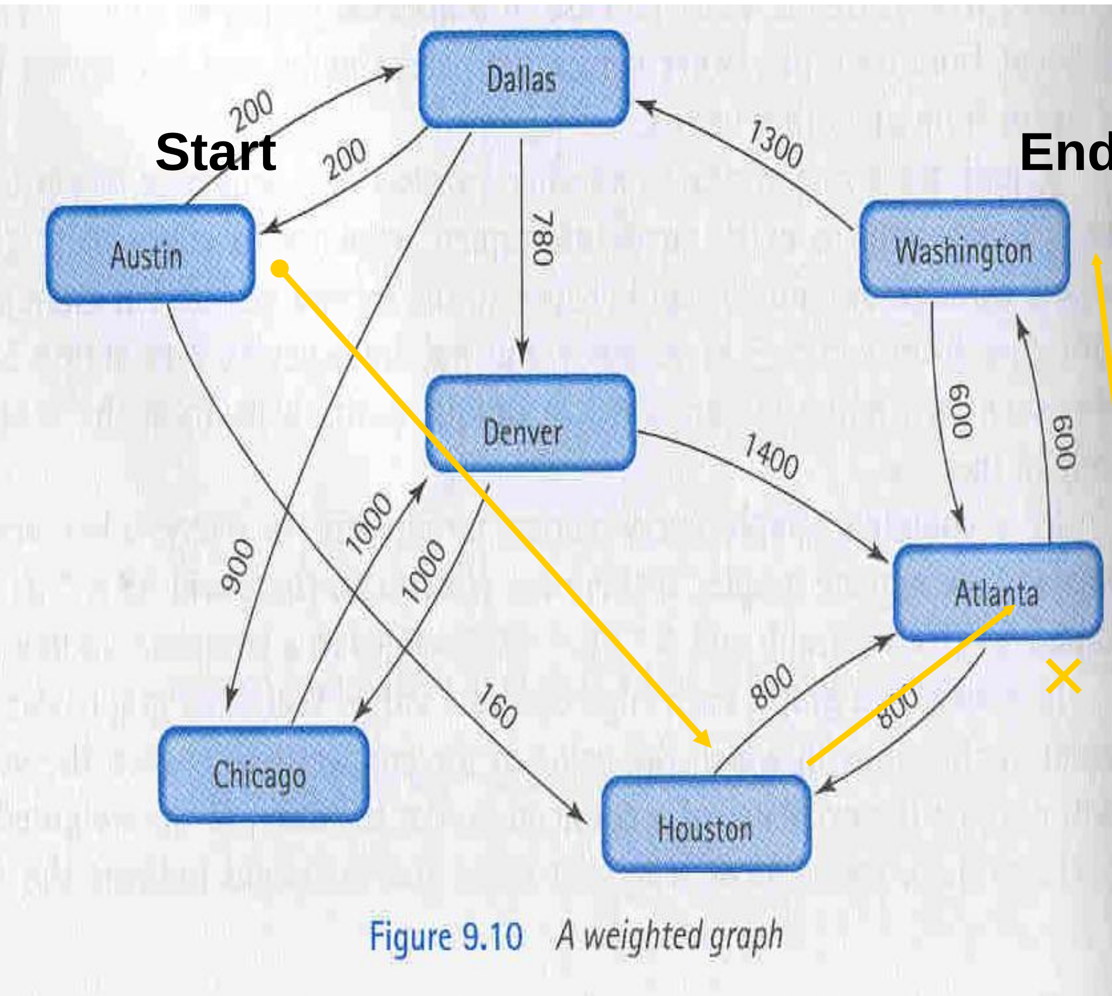


Figure 9.10 A weighted graph

Contents of the Stack

At Austin	Houston
	Dallas
At Houston	Atlanta
	Dallas
At Atlanta	Washington
	Dallas
At Washington	End

path:

Austin→Houston→Atlanta→ Washington

-Mark the vertex visited.

-Cycling

Additions to Graph ADT

- To avoid revisiting the vertex already visited, use a flag showing the status of visiting.

ClearMark

Function: initialize the flag for all vertices

MarkVertex(VertexType vertex)

Function: set the flag of vertex to True

Boolean IsMarked(VertexType vertex)

Function: return True if vertex is marked

```
template<class VertexType>
void DepthFirstSearch(GraphType
<VertexType> graph, VertexType
  startVertex,VertexType endVertex)
{
  using namespace std;
  StackType<VertexType> stack;
  QueType<VertexType> vertexQ;

  bool found = false;
  VertexType vertex;
  VertexType item;

  graph.ClearMarks();
  stack.Push(startVertex);
  do  {
    stack.Pop(vertex);
    if (vertex == endVertex)
    {
      cout << vertex);
      found = true;
    }
    else
    {
      if (!graph.IsMarked(vertex))
      {
        graph.MarkVertex(vertex);
        cout << vertex;
        graph.GetToVertices(vertex,
           vertexQ);
        while (!vertexQ.IsEmpty())
        {
          vertexQ.Dequeue(item);
          if (!graph.IsMarked(item))
            stack.Push(item);
        }
      }
    }
  }    while    (!stack.IsEmpty()    &&
   !found);
  if (!found)
    cout   <<   "Path   not   found."   <<
   endl;
}
```

Breadth-First Search(너비우선탐색)

Find all possible paths before move to the next level.

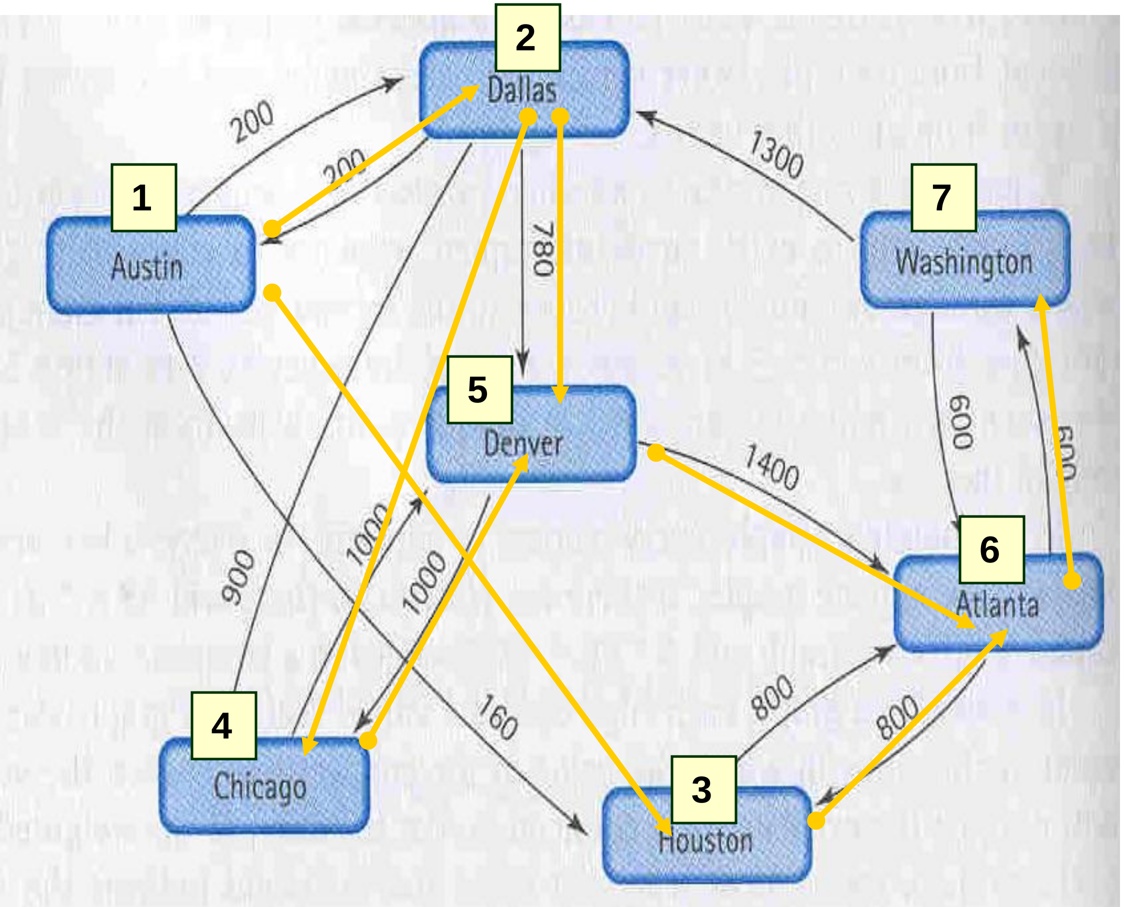


Figure 9.10 *A weighted graph*

	Contents of Queue
At Austin	Houston, Dallas
At Dallas	Denver, Chicago, ,Houston
At Houston	Atlanta, Denver , Chicago
At Chicago	Denver, Atlanta , Denver

Output:

Austin, Dallas, Houston, Chicago

```
void
   BreadthFirstSearch(GraphType<VertexType> graph,
   VertexType startVertex, VertexType endVertex)
{
  using namespace std;
  QueType<VertexType> queue;
  QueType<VertexType> vertexQ;
  bool found = false;
  VertexType vertex;
  VertexType item;
  graph.ClearMarks();
  queue.Enqueue(startVertex);
do
  {
    queue.Dequeue(vertex);

    if (vertex == endVertex)
    {
      cout  << vertex;
      found = true;
    }
else
    {
      if (!graph.IsMarked(vertex))
      {
        graph.MarkVertex(vertex);
        cout  << vertex;
        graph.GetToVertices(vertex, vertexQ);

        while (!vertexQ.IsEmpty())
        {
          vertexQ.Dequeue(item);
          if (!graph.IsMarked(item))
            queue.Enqueue(item);
        }
      }
    }
  } while (!queue.IsEmpty() && !found);
  if (!found)
  cout << "Path not found." << endl;
}
```

Single-Source Shortest-Path Problem

- Many paths may exist between two vertices
- Want to find the shortest path among the paths.

  Ex) Shortest path between Austin and Washington by Korean Airline

- Possible paths between Austin and Washington
  - Austin → Houston → Atlanta → Washington

    (160) + (800) + (600) = 1560
  - Austin→Dallas→Denver→Atlanta→Washington

    (200) + (780) + (1400) + (600) = 1980
- Shortest-Path Search
  - Use Priority Queue instead of queue in breadth first search algorithm. Take the vertex with minimum edge weight.

ShortestPath(graph,startVertex)

```
graph.ClearMarks()
item.fromVertex = startVertex
item.toVerex = startVertex
item.distance = 0
pq.Enqueue(item)
do
  pq.Dequeue(item)
  if item.toVertex is not Marked
    Mark item.fromVertex
    Write item
    item.fromVertex=item.toVertex
    minDistance = item.distance
    Item.fromVertex에 인접한 vertex
      들을 담은 Queue(vq)를 생성
    while !vq.Empty()
      vq.Dequeue(vertex)
      if !graph.IsMarkted(vertex)
        item.toVertex = vertex
        item.distance = minDistance
        +graph.WeightIs(fromVertex
          , vertex)
        pq.Enqueue(item)
while !pq.IsEmpty()
```

-Use min. heap for the Queue

-Continue until all the vertices are visited.

ShortestPath(graph,Washington)

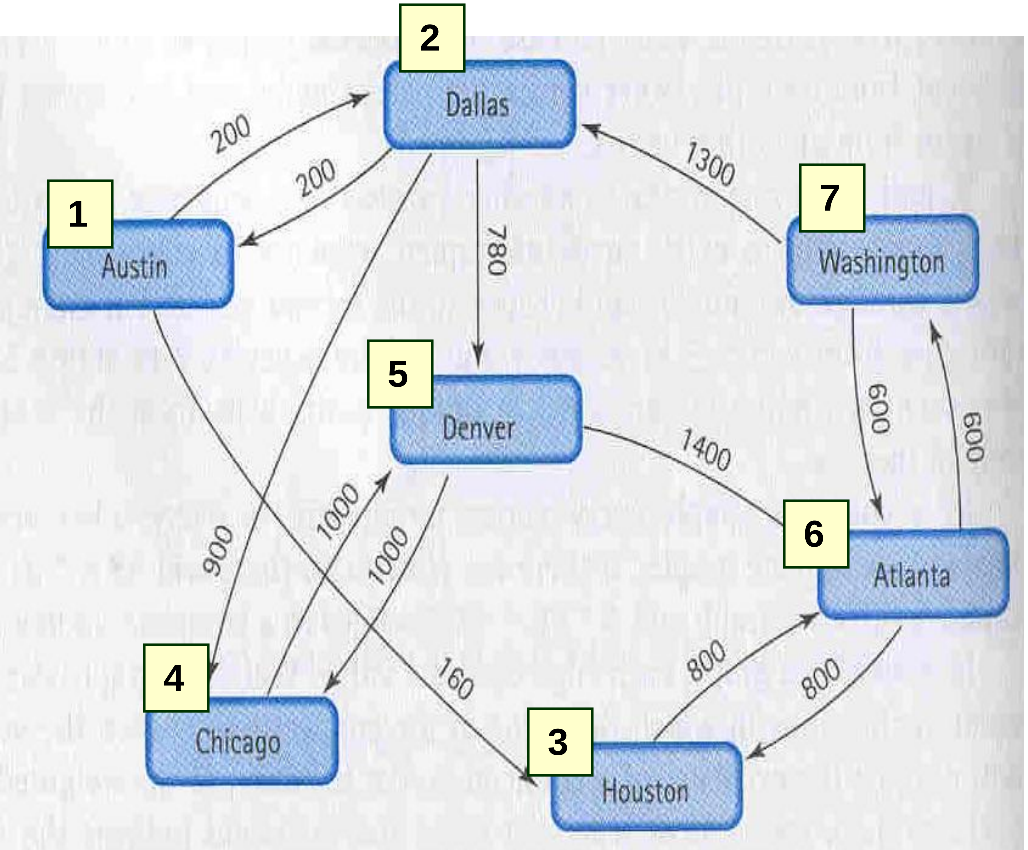


Figure 9.10 *A weighted graph*

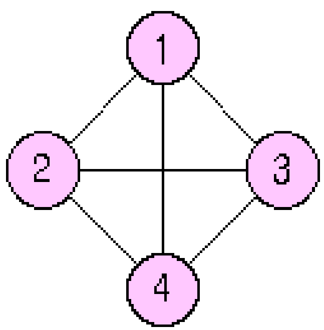
pQ: {(7,7,0)}
1. Washington Washington 0
vQ:{2(1300),6(600)}, mD=0
pQ: {(7,2,1300), (7,6,600)}
2. Washington Atlanta 600
vQ:{3(800)}, mD=600
pQ: {(6,3,1400),(7,2,1300)}
3. Washington Dallas 1300
vQ:{4(900),5(780),1(200)}, mD=1300
pQ: {(2,4,2200),(2,1,1500),
(6,3,1400) (2,5,2080),}
4. Atlanta Houston 1400
vQ:{}, mD=1400
pQ: {(2,4,2200),(2,5,2080)
,(2,1,1500)}
5. Dallas Austine 1500
6. Dallas Denver 2080
7. Dallas Chicago 2200

Find the minimum distance between Washing to and all other cities.

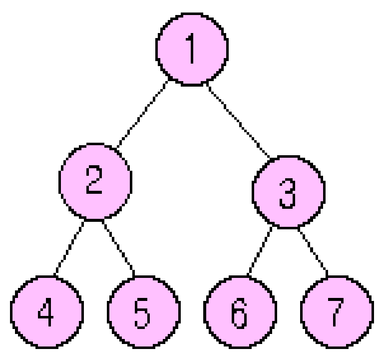
Array Implementation

- **Present the graph with N vertices by using a NxN Adjacency Matrix**
- **Adjacency Matrix: 2D array showing the connection between any two vertices. If the element (i,j) is 1, vertex i and j are connected. Otherwise, they not connected.**
- **Ex) Adjacency matrix of G1 Adjacency matrix of G3**

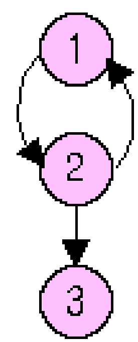
$$\begin{array}{c|c} & \begin{matrix} 1 & 2 & 3 & 4 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix} & \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix} \end{array} \qquad \begin{array}{c|c} & \begin{matrix} 1 & 2 & 3 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \end{matrix} & \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix} \end{array}$$



G_1



G_2



G_3

Graph of flight connections between cities

graph
.num Vertices 7
.vertices

[0]	"Atlanta "
[1]	"Austin "
[2]	"Chicago "
[3]	"Dallas "
[4]	"Denver "
[5]	"Houston "
[6]	"Washington"
[7]	
[8]	
[9]	

.edges

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
[0]	0	0	0	0	0	800	600	•	•	•
[1]	0	0	0	200	0	160	0	•	•	•
[2]	0	0	0	0	1000	0	0	•	•	•
[3]	0	200	900	0	780	0	0	•	•	•
[4]	1400	0	1000	0	0	0	0	•	•	•
[5]	800	0	0	0	0	0	0	•	•	•
[6]	600	0	0	1300	0	0	0	•	•	•
[7]	•	•	•	•	•	•	•	•	•	•
[8]	•	•	•	•	•	•	•	•	•	•
[9]	•	•	•	•	•	•	•	•	•	•

(Array positions marked '•' are undefined)

Figure 9.15 *Graph of flight connections between cities*

Linked Implementation

- Adjacency List: list containing adjacent vertices

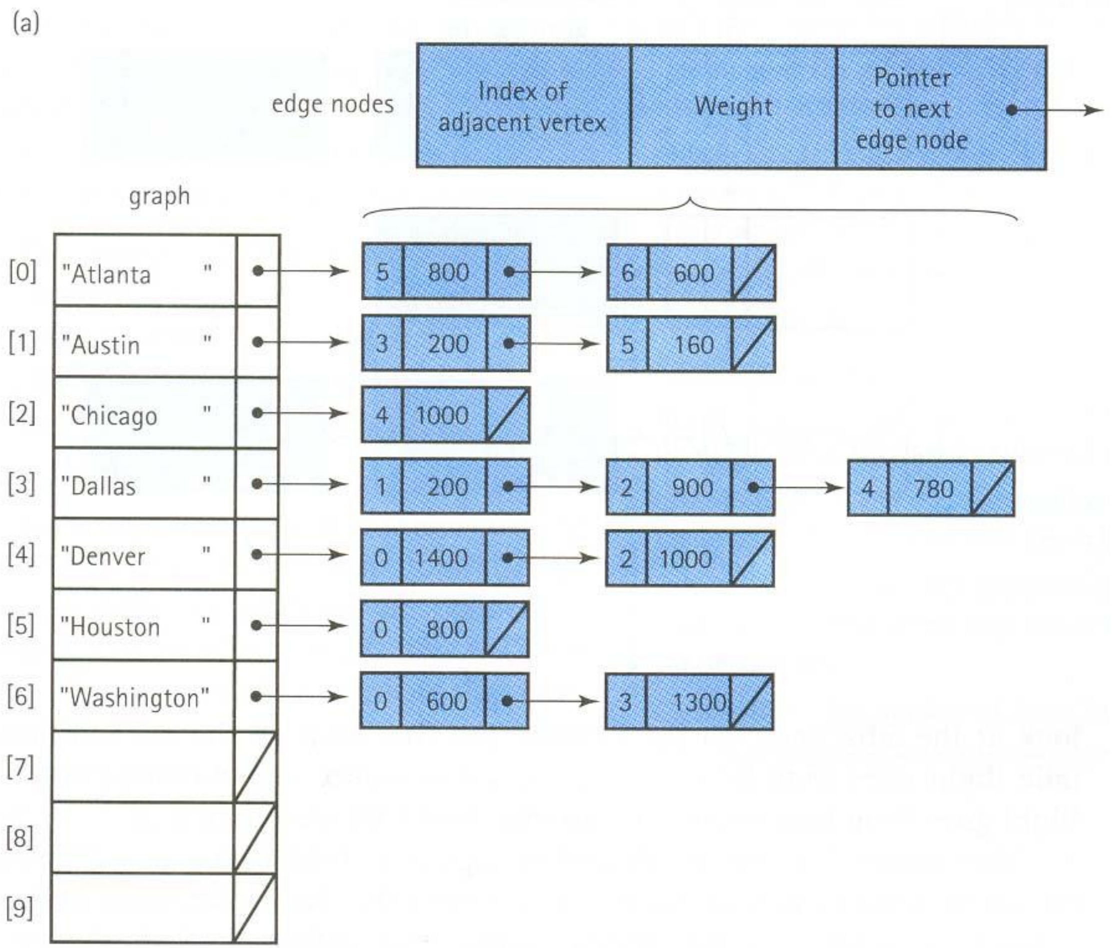


Figure 9.16 *Adjacency list representation of graphs*

Implementation using linked list only

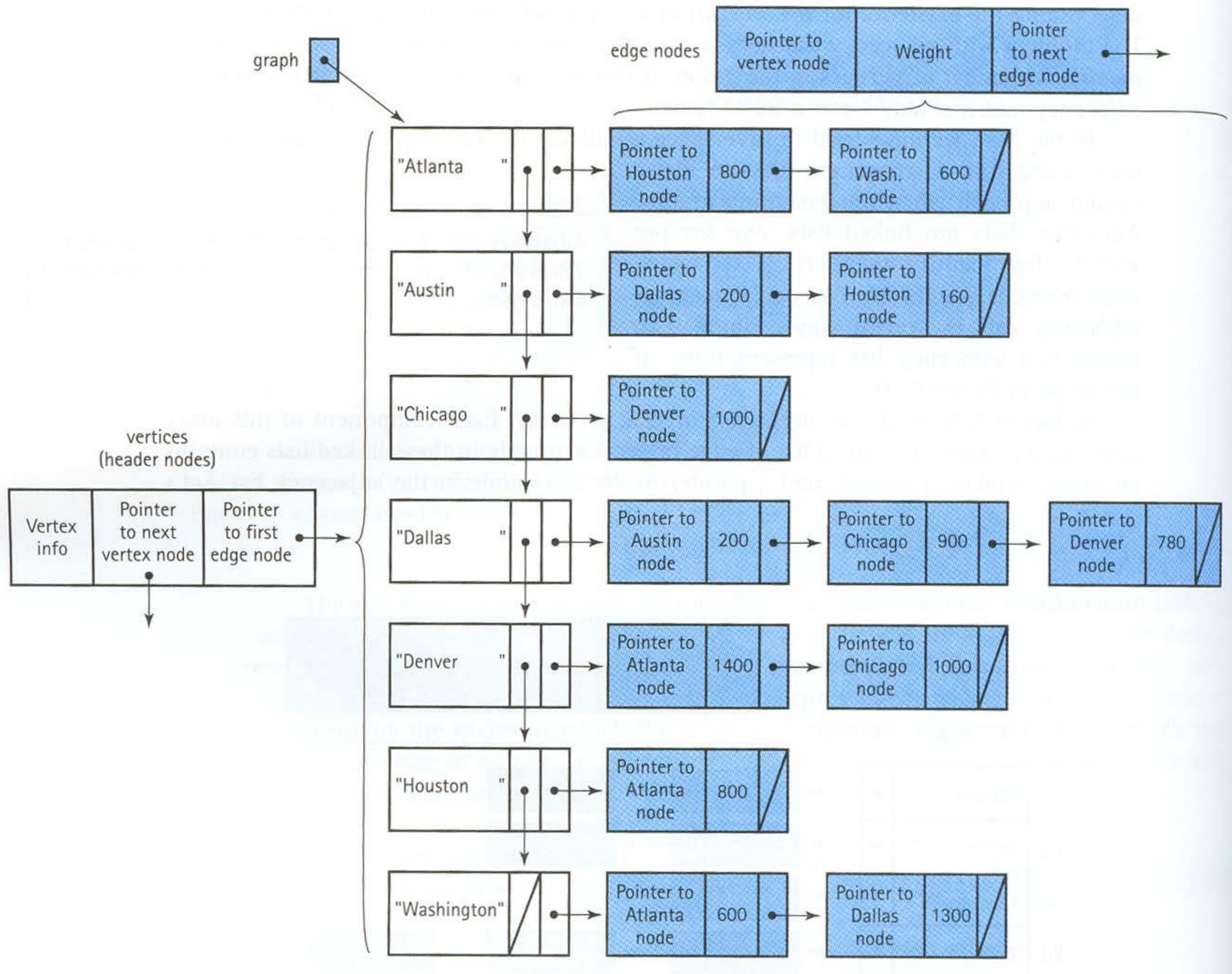


Figure 9.16 *(continued)*