Basic Data Structures

September 29, 2016

What are programs made of?

Programs= Data Structures + Algorithms

Data Abstraction

- Separation of a data type's logical properties from its implementation.
- Logical Properties
  - What are the possible values?
  - What operations will be needed?
- Implementation
  - How can this be done in C++?
  - How can data types be used?

Abstract Data Type (ADT)

- a data type that does not describe or belong to any specific data, yet allows the specification of organization and manipulation of data
- a data type whose properties (domain and operations) are specified independently of any particular implementation
- a data type that specifies and can share its logical properties without giving specifics of the implementation code
- a way of thinking about data types, often outside the constraints of a programming language
- actual data type is added later in an implementation

Four Basic Kinds of ADT Operations

- Constructor – creates a new instance (object) of an ADT.
- Transformer – changes the state of one or more of the data values of an instance.
- Observer – allows us to observe the state of one or more of the data values without changing them.
- Iterator – allows us to process all the components in a data structure sequentially.

What is a Stack?

- Logical level: A stack is an ordered group of homogeneous items (elements), in which the removal and addition of stack items can take place only at the top of the stack.
- A stack is a LIFO "last in, first out" structure.

Stack Operations

- MakeEmpty – Sets stack to an empty state.
- IsEmpty – Determines whether the stack is currently empty.
- IsFull – Determines whether the stack is currently full.
- Push (ItemType newItem) – Throws exception if stack is full; otherwise adds newItem to the top of the stack.
- Pop – Throws exception if stack is empty; otherwise removes the item at the top of the stack.
- ItemType Top – Throws exception if stack is empty; otherwise returns a copy of the top item

Class specification for Stack

```
// File StackType.h

class FullStack          // Exception class thrown by
                                    // Push when stack is full
{};
class EmptyStack                  // Exception class thrown by
                                         // Pop and Top when stack is empty
{};

#include "ItemType.h"

class StackType
{
public:

        StackType( );
        // Class constructor.
        bool IsFull () const;
        // Function: Determines whether the stack is full.
        // Pre: Stack has been initialized
        // Post: Function value = (stack is full)
```

Class specification for Stack (Cont.)

```
        bool IsEmpty() const;
        // Function: Determines whether the stack is empty.
        // Pre:   Stack has been initialized.
        // Post:  Function value = (stack is empty)
        void Push( ItemType newItem );
        // Function: Adds newItem to the top of the stack.
        // Pre: Stack has been initialized.
        // Post: If (stack is full), FullStack exception is thrown;
  //          otherwise, newItem is at the top of the stack.
  void Pop();
        // Function: Removes top item from the stack.
        // Pre: Stack has been initialized.
        // Post: If (stack is empty), EmptyStack exception is thrown;
  //          otherwise, top element has been removed from stack.
  ItemType Top();
        // Function: Returns a copy of top item on the stack.
        // Pre: Stack has been initialized.
        // Post: If (stack is empty), EmptyStack exception is thrown;
        //            otherwise, returns a copy of top element.
private:
        int top;
        ItemType  items[MAX_ITEMS];
};
```

Stack Implementation

```
// File: StackType.cpp

#include "StackType.h"
#include <iostream>
StackType::StackType( )
{
   top = -1;
}
bool StackType::IsEmpty() const
{
   return(top = = -1);
}

bool StackType::IsFull() const
{
   return (top = = MAX_ITEMS-1);
}
```

Stack Implementation (Cont.)

```
void StackType::Push(ItemType newItem)
{
        if( IsFull() )
           throw FullStack():
        top++;
        items[top] = newItem;
}

void StackType::Pop()
{
        if( IsEmpty() )
          throw EmptyStack();
   top--;
}

ItemType StackType::Top()
{
        if (IsEmpty())
          throw EmptyStack();
  return items[top];
}
```

Sample Stack Code

```
char     letter = 'V';
StackType  charStack;

charStack.Push(letter);

charStack.Push('C');

charStack.Push('S');

charStack.Pop();

charStack.Push('K');

while (!charStack.IsEmpty())
{
        letter = charStack.Top();
        charStack.Pop()
}
```

What is a Queue?

- Logical (or ADT) level: A queue is an ordered group of homogeneous items (elements), in which new elements are added at one end (the rear), and elements are removed from the other end (the front).
- A queue is a FIFO "first in, first out" structure.

Queue Operations

- MakeEmpty – Sets queue to an empty state.
- IsEmpty – Determines whether the queue is currently empty.
- IsFull – Determines whether the queue is currently full.
- Enqueue (ItemType newItem) – Adds newItem to the rear of the queue.
- Dequeue (ItemType& item) – Removes the item at the front of the queue and returns it in item.

Class specification for Queue

```
// Header file for Queue ADT.
class FullQueue
{};

class EmptyQueue
{};


struct NodeType;

#include "ItemType.h"

class QueType
{
public:
    QueType();
    // Class constructor.
    // Because there is a default constructor, the precondition
    // that the queue has been initialized is omitted.
    QueType(int max);
    // Parameterized class constructor.
    ~QueType();
    // Class destructor.
    void MakeEmpty();
    // Function: Initializes the queue to an empty state.
    // Post: Queue is empty.
```

Class specification for Queue

```
        bool IsEmpty() const;
        // Function: Determines whether the queue is empty.
        // Post: Function value = (queue is empty)
        bool IsFull() const;
        // Function: Determines whether the queue is full.
        // Post: Function value = (queue is full)
        void Enqueue(ItemType newItem);
        // Function: Adds newItem to the rear of the queue.
        // Post: If (queue is full) FullQueue exception is thrown
        //       else newItem is at rear of queue.
        void Dequeue(ItemType& item);
        // Function: Removes front item from the queue and returns it in item.
        // Post: If (queue is empty) EmptyQueue exception is thrown
        //       and item is undefined
        //       else front element has been removed from queue and
        //       item is a copy of removed element.
private:
        NodeType* front;
        NodeType* rear;
};

struct NodeType {
        ItemType info;
        NodeType* next;
};
```

Queue Implementation

```
QueType::QueType()          // Class constructor.
// Post:  front and rear are set to NULL.
{
  front = NULL;
  rear = NULL;
}


void QueType::MakeEmpty()
// Post: Queue is empty; all elements have been deallocated.
{
  NodeType* tempPtr;

  while (front != NULL)
  {
    tempPtr = front;
    front = front->next;
    delete tempPtr;
  }
  rear = NULL;
}

          // Class destructor.
QueType::~QueType()
{
  MakeEmpty();
}
```

Queue Implementation (Cont.)

```
bool QueType::IsFull() const
{
  NodeType* location;
  try
  {
    location = new NodeType;
    delete location;
    return false;
  }
  catch(bad_alloc exception)
  {
    return true;
  }
}


bool QueType::IsEmpty() const
{
  return (front == NULL);
}


void QueType::Enqueue(ItemType newItem)
{
  if (IsFull())
    throw FullQueue();
  else
  {
    NodeType* newNode;

    newNode = new NodeType;
    newNode->info = newItem;
    newNode->next = NULL;
    if (rear == NULL)
      front = newNode;
    else
      rear->next = newNode;
    rear = newNode;
  }
}
```

Queue Implementation (Cont.)

```
void QueType::Dequeue(ItemType& item)
// Removes front item from the queue and returns it in item.
// Pre:  Queue has been initialized and is not empty.
// Post: If (queue is not empty) the front of the queue has been
//       removed and a copy returned in item;
//       othersiwe a EmptyQueue exception has been thrown.
{
  if (IsEmpty())
    throw EmptyQueue();
  else
  {
    NodeType* tempPtr;

    tempPtr = front;
    item = front->info;
    front = front->next;
    if (front == NULL)
      rear = NULL;
    delete tempPtr;
  }
}
```

Sample Queue Code

```
/ Test driver
#include <iostream>
#include "QueType.h"
using namespace std;

int main()
{

  ifstream inFile;        // file containing operations
  ofstream outFile;       // file containing output
  string inFileName;      // input file external name
  string outFileName;     // output file external name
  string outputLabel;
  string command;         // operation to be executed

  ItemType item;
  QueType queue;
  int numCommands;


  // Prompt for file names, read file names, and prepare files
  cout << "Enter name of input command file; press return." << endl;
  cin  >> inFileName;
  inFile.open(inFileName.c_str());

  cout << "Enter name of output file; press return." << endl;
  cin  >> outFileName;
  outFile.open(outFileName.c_str());

  cout << "Enter name of test run; press return." << endl;
  cin  >> outputLabel;
  outFile << outputLabel << endl;

  inFile >> command;
```

Sample Queue Code

```
numCommands = 0;
 while (command != "Quit")
 {
   try
   {
     if (command == "Enqueue")
     {
       inFile >> item;
       queue.Enqueue(item);
       outFile << item << " is enqueued." << endl;
     }
     else if (command == "Dequeue")
     {
       queue.Dequeue(item);
       outFile<< item  << " is dequeued. " << endl;
     }
     else if (command == "IsEmpty")
       if (queue.IsEmpty())
         outFile << "Queue is empty." << endl;
       else
         outFile << "Queue is not empty." << endl;

     else if (command == "IsFull")
       if (queue.IsFull())
         outFile << "Queue is full." << endl;
       else outFile << "Queue is not full."  << endl;
   }
   catch (FullQueue)
   {
     outFile << "FullQueue exception thrown." << endl;
   }

   catch (EmptyQueue)
   {
     outFile << "EmtpyQueue exception thrown." << endl;
   }
   numCommands++;
   cout <<  " Command number " << numCommands << " completed."
        << endl;
   inFile >> command;

 };

 cout << "Testing completed."  << endl;
 inFile.close();
 outFile.close();
 return 0;
}
```

Tree

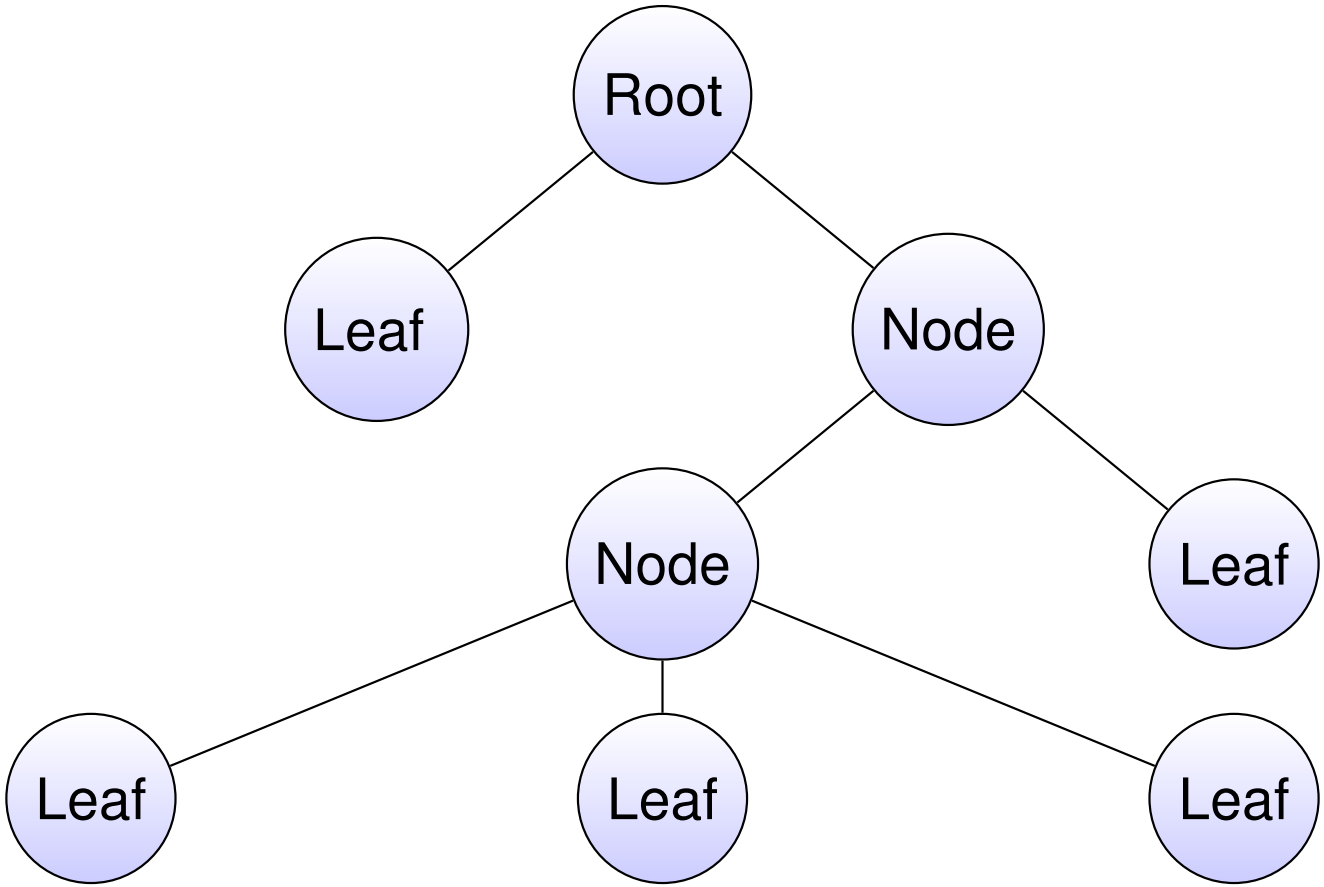
A tree is a finite set of one or more nodes such that:

- There is a specially designated node called the root.
- The remaining nodes are partitioned into $n \geq 0$ disjoint sets $T_1, ..., T_n$, where each of these sets is a tree.
- We call $T_1, ..., T_n$ the subtrees of the root.

Terminology

- The degree of a node is the number of subtrees of the node
- The node with degree 0 is a leaf or terminal node.
- A node that has subtrees is the parent of the roots of the subtrees.
- The roots of these subtrees are the children of the node.
- Children of the same parent are siblings.
- The ancestors of a node are all the nodes along the path from the root to the node.

Tree Notation

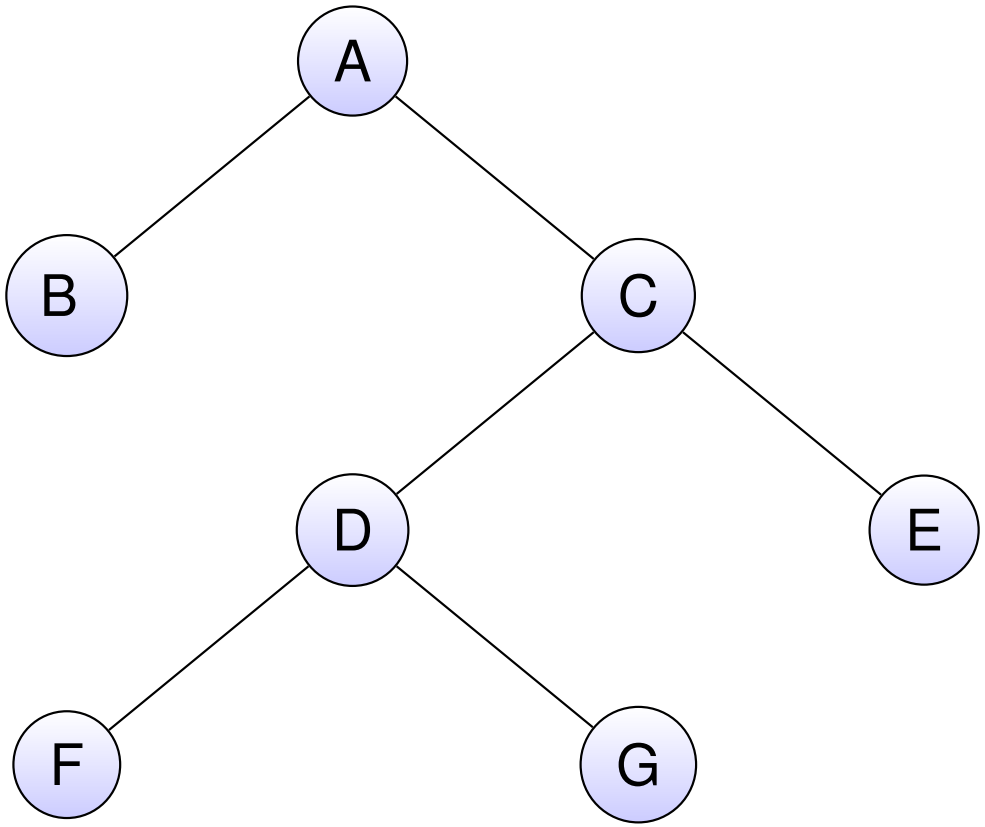


Binary Tree

A binary tree is a structure in which:

- Each node can have at most two children, and in which a unique path exists from the root to every other node.
- The two children of a node are called the left child and the right child,if they exist.

Binary Tree Example



Linked Representation

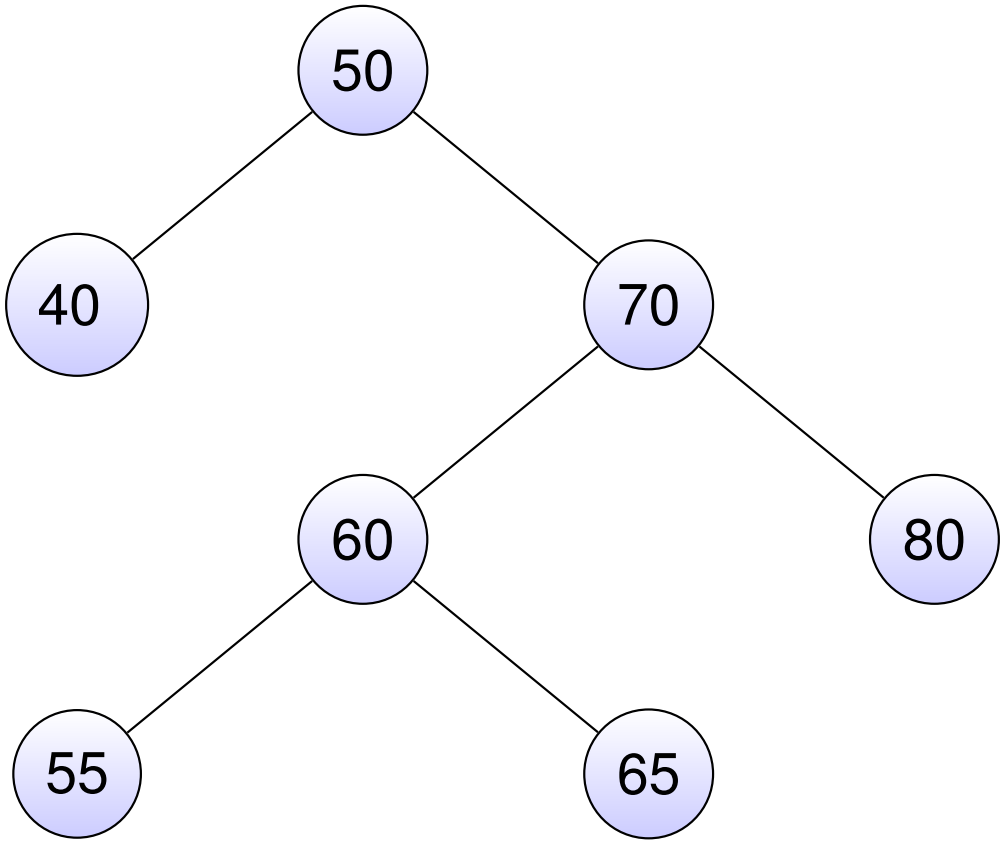
```
typedef struct node *tree_pointer;
typedef struct node {
        int info;
        tree_pointer left_child, right_child;
};
```

Binary Search Tree

A special kind of binary tree in which:

- Each node contains a distinct data value,
- The key values in the tree can be compared using "greater than" and "less than", and
- The key value of each node in the tree is less than every key value in its right subtree, and greater than every key value in its left subtree

Binary Search Tree Example



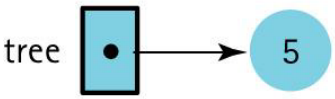
Insertion into a Binary Search Tree

```
void Insert(TreeNode*& tree, ItemType item)
{
        if (tree == NULL)
        {// Insertion place found.
                tree = new TreeNode;
                tree->right = NULL;
                tree->left = NULL;
                tree->info = item;
        }
        else if (item < tree->info)
                Insert(tree->left, item);
        else
                Insert(tree->right, item);
}
```

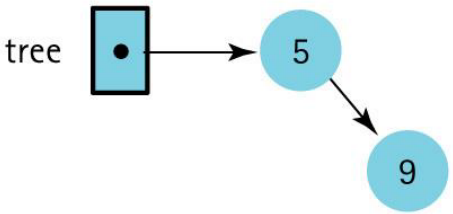
Binary Search Tree Insertion Example

(a) tree

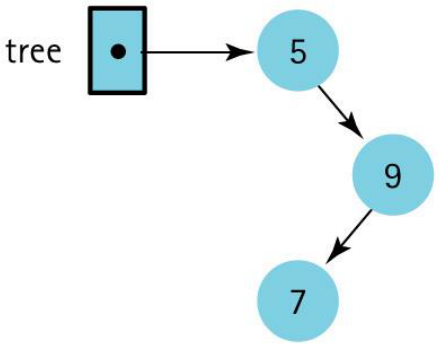
(b) Insert 5



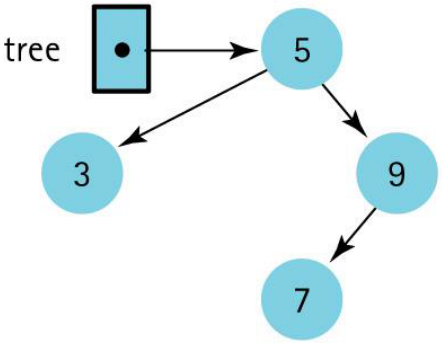
(c) Insert 9



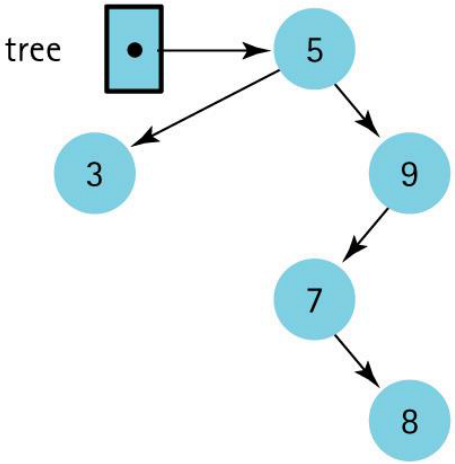
(d) Insert 7



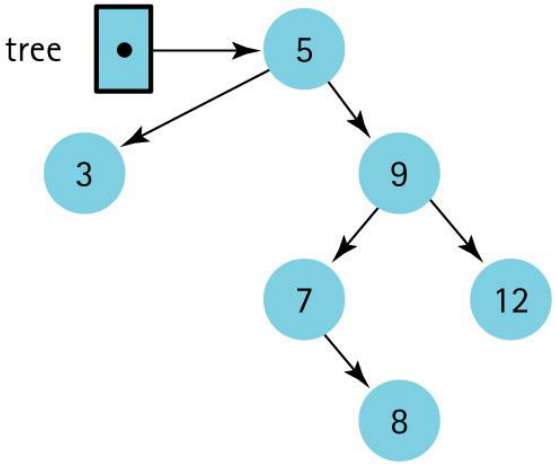
(e) Insert 3



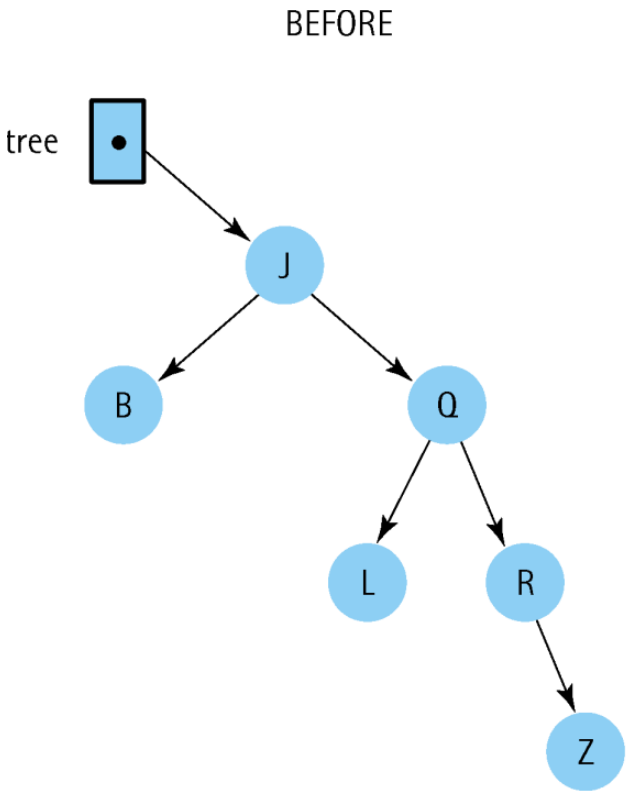
(f) Insert 8

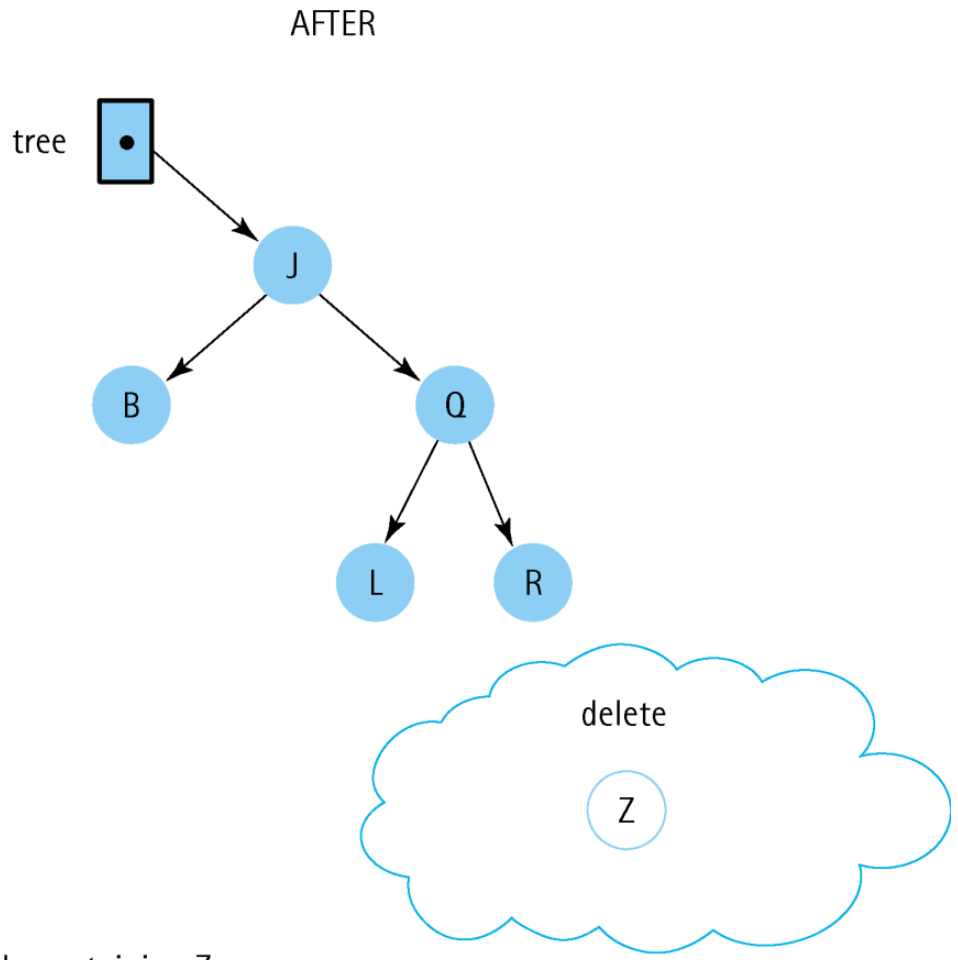


(g) Insert 12



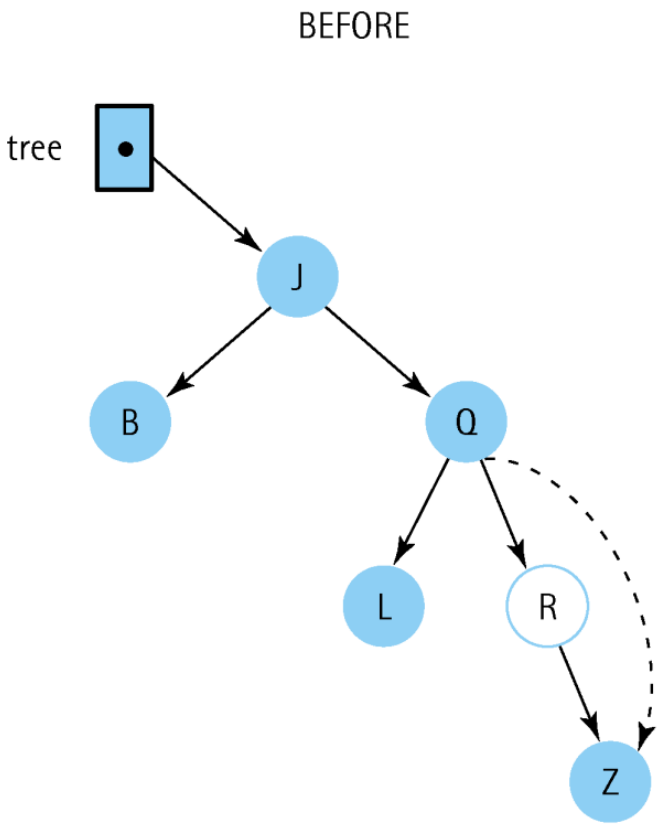
Deleting a Leaf Node

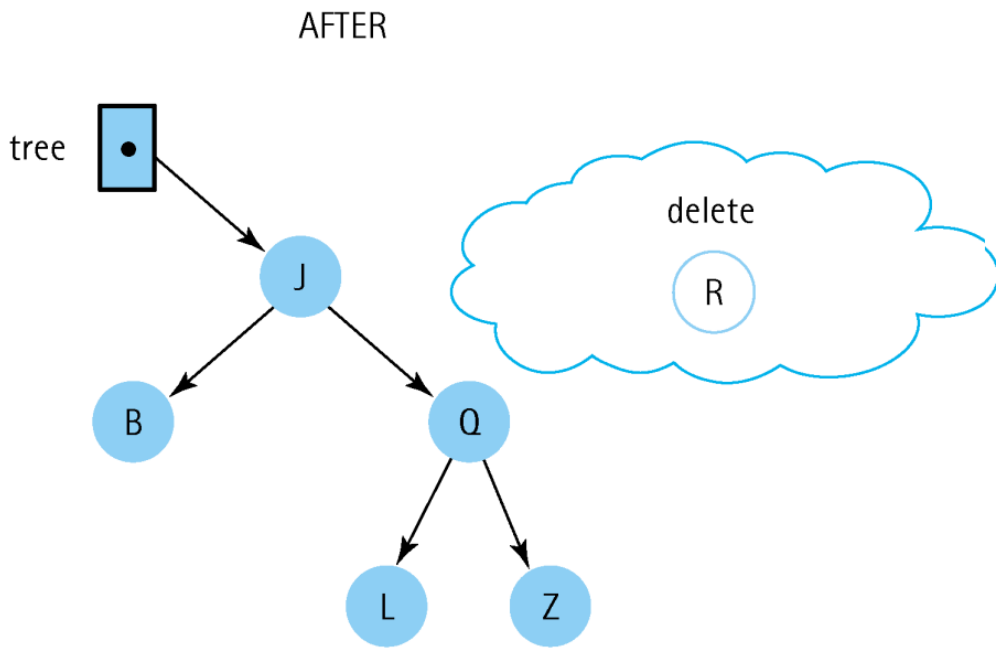




Delete the node containing Z

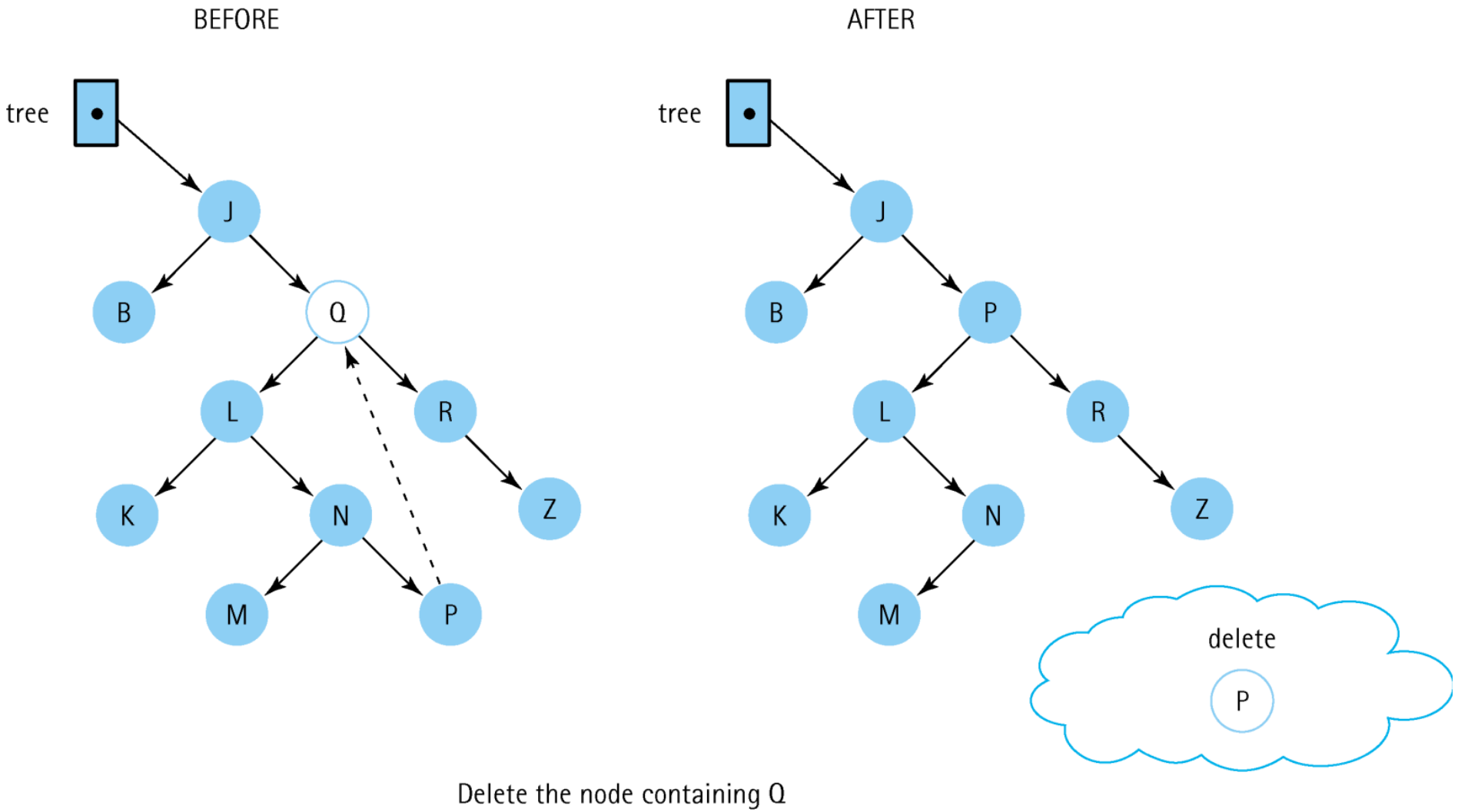
Deleting a Node with One Child





Delete the node containing R

Deleting a Node with Two Children



Delete the node containing Q

Delete

```
void Delete(TreeNode*& tree, ItemType item)
{
        if (item < tree->info)
                Delete(tree->left, item);
        else if (item > tree->info)
                Delete(tree->right, item);
        else
                DeleteNode(tree); // Node found
}
```

DeleteNode

```
void DeleteNode(TreeNode*& tree)
{
        ItemType data;
        TreeNode* tempPtr;
        tempPtr = tree;
        if (tree->left == NULL) {
                tree = tree->right;
                delete tempPtr; }
        else if (tree->right == NULL){
                tree = tree->left;
                delete tempPtr;}
        else
        {
                GetPredecessor(tree->left, data);
                tree->info = data;
                Delete(tree->left, data);
        }
}
```

Get Predecessor

```
void GetPredecessor(TreeNode* tree, ItemType& data)
{
        while (tree->right != NULL)
                tree = tree->right;
        data = tree->info;
}
```