Binary Trees and Hash Tables

Binary Trees

An Example of a Binary Tree

A binary tree node is a structure variable or an object with three types of member variables:

- A data part: one or more member variables appropriate to hold the data in the node.
- A left part: a pointer variable to hold the address of the left child node.
- A right part: a pointer variable to hold the address of the right child node.

The left part of a node is set to NULL if that node has no left child and its right part is set to NULL if it has no right child.

The first node of a binary tree is called the root node.

The address of the root node is stored into a pointer variable known as root.

The root of an empty tree is set to NULL.

A node (other than the root) is the (left/right) child of only one node.

A leaf node is a node without a (left/right) child.

In the tree above, we have the following:

- The root node is the node with data value 40.
- Its left child is the node with data value 20; and its right child is the node with data value 10.
- The node with data value 20 has no right child.
- The leaf nodes are the nodes with data values 60, 30 and 70.
You declare a binary tree node in the same way that you declare a linked list node.

For example, a node of the binary tree in the example above can be defined from the following structure:

```c
struct TreeNode
{
    int value; //data part
    TreeNode *leftChild; // left child address
    TreeNode *rightChild; // right child address
};
```

The node to hold the information about a product represented by the following structure type:

```c
struct ProductInfo
{
    int prodNum; // to hold a product number
    double unitPrice; // to hold a product unit price
    int quantity; // to hold a product quantity
};
```

Can be defined by one of the following structures:

```c
struct ProductNode1
{
    int prodNum; // to hold a product number
    double unitPrice; // to hold its unit price
    int quantity; // to hold its quantity
    ProductNode1 *Lchild; // address of left child
    ProductNode1 *Rchild; // address of right child
};
```

```c
struct ProductNode2
{
    ProductInfo product; // a product’s information
    ProductNode2 *Lchild; // address of left child
    ProductNode2 *Rchild; // address of right child
};
```

**Examples of Nodes Declared as Object Instances of a Class**

a. Node to hold an integer value:

```c
class TreeNode
{
    Public:
    int value; //data part
    TreeNode *leftChild; // left child address
    TreeNode *rightChild; // right child address
};
```
b. The node to hold the information about a product represented by the following structure type:

```c
struct ProductInfo
{
    int prodNum;    // to hold a product number
    double unitPrice; // to hold a product unit price
    int quantity;    // to hold a product quantity
};
```

Can be defined by one of the following classes:

```c
class ProductNode1
{
    public:
        int prodNum;    // to hold a product number
        double unitPrice; // to hold its unit price
        int quantity;    // to hold its quantity
        ProductNode1 *Lchild; // address of left child
        ProductNode1 *Rchild; // address of right child
};
```

```c
class ProductNode2
{
    public:
        ProductInfo product; // a product’s information
        ProductNode2 *Lchild; // address of left child
        ProductNode2 *Rchild; // address of right child
};
```

In general, given a class \textit{AClass}, we can define a class that can be used to create nodes with its data members as the data part as follows:

```c
class AClassNode
{
    public:
        AClass data;
        AClassNode *leftChild;
        AClassNode *rightChild;
};
```

This class may also have constructors and other member functions.

- You create an empty binary tree by setting its root to \texttt{NULL}.

\textbf{Examples}

```c
TreeNode *bRoot = NULL;
ProductNode2 *productRoot = NULL;
```
Exercise B1

1. Write the definition of a structure and the definition of a class that can be used to create a node of a binary tree with a double precision value in its data part.

2. The information about a course is represented by the following structure:

   ```c
   struct CourseInfo
   {
       string courseNum;
       int credit;
       double gpv;
   };
   ```

   Write the definition of a structure and the definition of a class that can be used to create a node of a binary tree with this information in its data part.

3. Write the definition of a class that can be used to create a node of a binary tree with the data part the data members of the class Data that you defined in exercise O7.

Binary Tree Traversal

- A binary tree is traversed from the root node to the leaf nodes in one of the following orders:

A. **In-order (LVR) traversal**
   1. Traverse the leftmost sub-tree of a node.
   2. Visit the node.
   3. Traverse the rightmost sub-tree.

   For the tree above, the in-order traversal visits the nodes as follows: 60, 20, 40, 30, 10, 70.

B. **Pre-order (VLR) traversal**
   1. Visit the node
   2. Traverse the leftmost sub-tree.
   3. Traverse the rightmost sub-tree

   For the tree above, the pre-order traversal visits the nodes as follows: 40, 20, 60, 10, 30, 70.

C. **Post-order (LRV) traversal**
   1. Traverse the leftmost sub-tree.
   2. Traverse the rightmost sub-tree
   3. Visit the node.

   For the tree above, the post-order traversal visits the nodes as follows: 60, 20, 30, 70, 10, 40.
Given that the following structure is used for the definition of a node of a binary tree, we provide the recursive definition of a function to output the data part of the nodes of a binary tree for each of the above binary tree traversal as follows:

```c
typedef int   dataType;
struct TreeNode
{
    dataType data;       //data part
    TreeNode *leftChild; // left child address
    TreeNode *rightChild; // right child address
};

/*-----------------------------inorderTraversal( )----------------------------------------*/
/* output the data part of the nodes of a binary tree in the LVR order */
void   inorderTraversal( ostream & out,  TreeNode * subtreeRoot )
{
    if( subtreeRoot ) // subtreeRoot != NULL
    {
        inorderTraversal( out,  subtreeRoot->leftChild );
        out    <<    subtreeRoot->data;
        inorderTraversal( out,  subtreeRoot->rightChild );
    } // do nothing if the tree is empty
}

/*-----------------------------preorderTraversal( )----------------------------------------*/
/* output the data part of the nodes of a binary tree in the VLR order */
void   preorderTraversal( ostream & out,  TreeNode * subtreeRoot )
{
    if( subtreeRoot ) // subtreeRoot != NULL
    {
        out    <<    subtreeRoot->data;
        preorderTraversal( out,  subtreeRoot->leftChild );
        preorderTraversal( out,  subtreeRoot->rightChild );
    } // do nothing if the tree is empty
}
```
```cpp
/*-------------------------postorderTraversal( )-------------------------*/
/* output the data part of the nodes of a binary tree in the LRV order */

void postorderTraversal( ostream & out, TreeNode * subtreeRoot )
{
    if( subtreeRoot ) // subtreeRoot != NULL
    {
        postorderTraversal( out, subtreeRoot -> leftChild );
        postorderTraversal( out, subtreeRoot -> rightChild );
        out << subtreeRoot -> data;
    } // do nothing if the tree is empty
}

Exercise B2

Given the following binary tree:

1. Output the data part of the nodes of this tree in the in-order (LVR) traversal of the binary tree.
2. Output the data part of the nodes of this tree in the pre-order (VLR) traversal of this binary tree.
3. Output the data part of the nodes of this tree in the post-order (LRV) traversal of this binary tree.

Exercise B3 (extra credits)

1. A pre-order traversal of a binary tree has produced: A D F G H K L P Q R W Z, and an in-order traversal has produced: G F H K D L A W R Q P Z. Draw that binary tree.
2. Draw two different binary trees with the same pre-order and the same post-order traversals.
Binary Search Tree

- A **binary search tree** is a binary tree with the following characteristic:
  
  For each node in the tree,
  - The value at each node in its left sub-tree is smaller than the value at that node, and
  - The value at each node in its right sub-tree is greater than the value at that node.

**Examples of (non) Binary Search Trees**

The following tree is a binary search tree:

The following tree is not a binary search tree because 11 is the value of a node in the left sub-tree of the node with value 9.
The following tree is not a binary search tree because 14 is the value of a node in the right sub-tree of the node with value 16.

![Binary Search Tree Example](image)

**Exercise B4**

Which of the following trees are not binary search trees? Justify your answers.

a. 

![Binary Search Tree Example](image)

b. 

![Binary Search Tree Example](image)

c. 

![Binary Search Tree Example](image)
Basic Operations on a Binary Search Tree

- The following are the basic operations on a binary search tree:
  - Create an empty binary search tree (BST)
  - Determine if a binary search tree is empty.
  - Search a binary search tree for a given value.
  - Insert an item in the binary search tree.
  - Delete an item from a binary search tree
  - Traverse a binary search tree.

- In order to define these basic operations, we use the following structure for the definition of a node of a BST:

```c
typedef int dataType;
struct TreeNode
{
    dataType data;          // data part
    TreeNode *leftChild;    // left child address
    TreeNode *rightChild;   // right child address
};
```

Creating an Empty BST

- You create an empty BST by setting its root to NULL.

  Example:  
  ```
  TreeNode * bstRoot  = NULL;
  ```

Determining if a BST is Empty

- You determine if a BST is empty by checking if its root is set to NULL.

  Example:  
  ```
  bstRoot  == NULL;
  ```

Traversing a BST

- The nodes of a BST can be visited in the in-order (LVR), pre-order (VLR), or the post-order (LRV) traversal discussed above.
Note that for the in-order (LVR) traversal of a BST, the values of the nodes are listed in ascending order.

**Example:**

The values of the nodes of the BST in the example above are listed in the in-order (LVR) traversal as follows: 5 6 7 8 9 10 11 12 14 17 18 19 20

They are listed in the pre-order (VLR) traversal as follows: 10 8 7 5 6 9 20 14 12 11 18 17 19

They are listed in the post-order (LRV) traversal as follows: 6 5 7 9 8 11 12 17 19 18 14 20

inorderTraversal(  ) function defined above to perform an in-order traversal of a tree is a recursive function. That means that it must be called with the name of the root of the tree as an argument. Therefore, if it is declared as a member function of a class that is used to create BST objects, you will have to know the name of the root of the BST in order to call this function; which is not possible because the name of the root of a BST is a private data member of the class.

This issue is resolved by defining a public member function of a class (to create BST objects with the name of the root bstRoot ) as follows:

```cpp
void performInOrder( ostream & out )
{
    inorderTraversal( out, bstRoot );
}
```

And declaring inorderTraversal(  ) (defined below) as a private member function of the class.

```cpp
/*--------------------------inorderTraversal(  )----------------------------------------*/
/* output the data part of the nodes of a binary tree in the LVR order */
void inorderTraversal( ostream & out, TreeNode * subtreeRoot )
{
    if( subtreeRoot ) // subtreeRoot != NULL
    {
        inorderTraversal( out, subtreeRoot->leftChild );
        out << subtreeRoot->data;
        inorderTraversal( out, subtreeRoot->rightChild );
    } // do nothing if the tree is empty
}
```

The pre-order traversal and the post-order traversal operations are defined in a similar way.
Searching a BST

- You search a BST for the given value in the variable \textit{item} as follows:

1. Set the variable \textit{found} to false.
2. Set the pointer variable \textit{nodePtr} to the root of the tree: \textit{nodePtr} = \textit{bstRoot}.
3. While \textit{found} is false and \textit{nodePtr} is not NULL, do the following
   a. if \textit{item} = \textit{nodePtr} -> data, set variable \textit{found} to true.
   b. Otherwise, if \textit{item} < \textit{nodePtr} -> data, make the leftChild node of the current node the current node: \textit{nodePtr} = \textit{nodePtr} -> leftChild.
   c. Otherwise (the given value is greater than the value at the current node) make the rightChild node the current node: \textit{nodePtr} = \textit{nodePtr} -> rightChild.
4. Return \textit{found}.

Example

Given the following binary search tree:

![Binary search tree diagram]

a. We search for the value 18 in this BST as follows:
   1. We compare 18 to 10: because it is greater, we continue with the rightmost sub-tree.
   2. We compare 18 to 20: because it is smaller we continue with the leftmost sub-tree.
   3. We compare 18 to 14: because it is greater, we continue with the rightmost sub-tree.
   4. We compare 18 to 18: and it is found.

b. We search for the value 13 in this BST as follows:
   1. We compare 13 to 10: because it is greater, we continue with the rightmost sub-tree.
   2. We compare 13 to 20: because it is smaller we continue with the leftmost sub-tree.
   3. We compare 13 to 14: because it is smaller, we continue with the leftmost sub-tree.
   4. We compare 13 to 12: because it is greater, we continue with the rightmost sub-tree.
   5. The rightmost tree is empty. So, 13 is not in the BST.
The corresponding member function is written as follows:

```c
bool searchBST( dataType item )
{
    bool found = false;
    TreeNode * nodePtr = bstRoot;    // start at the root of the tree
    while ( !found && nodePtr )      // nodePtr != NULL
    {
        if ( item == nodePtr->data )  // the item is found in the tree
            found = true;
        else if ( item < nodePtr->data )
            nodePtr = nodePtr->leftChild;    // search the left sub-tree
        else                      // item is greater than the data at this node
            nodePtr = nodePtr->rightChild;   // search the right sub-tree
    }
    return found;
}
```

**Exercise B5**

Write the steps in the search of the values 2 and 17 in the BST of the example above.

**Inserting a Value into a BST**

- A value is inserted into a BST only if it is not already in the tree: we must therefore search the BST to make sure that it is not already there before we can insert it.
- If the BST is empty, the value is inserted at the root node of the BST.
- Otherwise, it is inserted at a leaf node (if it is not already in the BST).

**Example**

![Diagram of BST insertion](image-url)
The value 8 is inserted into the BST as follows:

1. We compare 8 to 12: because it is smaller, we continue with the leftmost sub-tree.
2. We compare 8 to 7: because it is greater we continue with the rightmost sub-tree.
3. We compare 8 to 9: because it is smaller, we continue with the leftmost sub-tree.
4. The leftmost sub-tree is empty: 8 is inserted in the leftChild node of the node with value 9.

The algorithm follows:

1. If \( bstRoot = NULL \), (The BST is empty) do the following
   a. Create a new node and make it the root node: \( bstRoot = \text{new TreeNode} \);
   b. Store the value to be inserted into the new node: \( bstRoot \rightarrow data = item \);
   c. Set the \( leftChild \) and the \( rightChild \) pointers of the new node to \( \text{NULL} \)
      \( bstRoot \rightarrow leftChild = bstRoot \rightarrow rightChild = \text{NULL} \);
2. Otherwise, do the following:
   a. Set the pointer variable \( nodePtr \) to the root of the tree: \( nodePtr = bstRoot \);
   b. Set the Boolean variable \( found \) to false.
   c. If( \( item = nodePtr \rightarrow data \) ) Set the Boolean variable \( found \) to true.
   d. Otherwise, if(\( item < nodePtr \rightarrow data \) ) set the pointer variable \( nextPtr \) to the \( leftChild \)
      pointer of the current node: \( nextPtr = nodePtr \rightarrow leftChild \);
   e. Otherwise, set the pointer variable \( nextPtr \) to the \( rightChild \) pointer of the current node:
      \( nextPtr = nodePtr \rightarrow rightChild \);
   f. While \( found \) is false and \( nextPtr \) is not \( \text{NULL} \), do the following
      I. Make the next pointer the current pointer: \( nodePtr = nextPtr \);
      II. If( \( item = nodePtr \rightarrow data \) ) Set the Boolean variable \( found \) to true.
      III. Otherwise, if(\( item < nodePtr \rightarrow data \) ) set the pointer variable \( nextPtr \) to the \( leftChild \)
           pointer of the current node: \( nextPtr = nodePtr \rightarrow leftChild \);
      IV. Otherwise, set the pointer variable \( nextPtr \) to the \( rightChild \) pointer of the current node:
           \( nextPtr = nodePtr \rightarrow rightChild \);
   g. If \( (found = false) \) do the following: (the value is not in the tree)
      I. Create a new node and store its address in the pointer variable \( newPtr \):
         \( \text{TreeNode} \*newPtr = \text{new TreeNode} \);
      II. Store the value to be inserted into the new node: \( newPtr \rightarrow data = item \);
      III. Set the \( leftChild \) and the \( rightChild \) pointers of the new node to \( \text{NULL} \):
         \( newPtr \rightarrow leftChild = newPtr \rightarrow rightChild = \text{NULL} \);
      IV. Make the new node the \( leftChild/rightchild \) of the current node:
         if(\( item < nodePtr \rightarrow data \) ), \( nodePtr \rightarrow leftChild = newPtr \);
         otherwise, \( nodePtr \rightarrow rightChild = newPtr \);
The corresponding member function is written as follows:

```cpp
void insertBST( dataType item )
{
  /*--------------------------- make the new node the root node if the BST is empty --------*/
  if ( !bstRoot ) // the BST is empty
  {
    bstRoot = new TreeNode;
    bstRoot -> data = item;
    bstRoot -> leftChild = bstRoot -> rightChild = NULL;
  }
  else // find out if the value is already in the tree;
  {
    // if it is not in the tree, look for the parent node of the new node
    TreeNode *nodePtr = bstRoot, // to hold the address of the current node
    *nextPtr; // to hold the address of the left/right child of the current node
    bool found = false;
    /*---------------- compare the given value with the value at the root node --------*/
    if( item == nodePtr -> data ) // the item is found in the tree
      found = true;
    else if( item < nodePtr -> data )
      nextPtr = nodePtr -> leftChild; // make the leftmost sub-tree the next sub-tree
    else
      nextPtr = nodePtr -> rightChild; // make the rightmost sub-tree the next sub-tree
    while ( !found && nextPtr ) // nextPtr != NULL
      {
        nodePtr = nextPtr; // make the next sub-tree the current sub-tree
        if( item == nodePtr -> data ) // the item is found in the sub-tree
          found = true;
        else if( item < nodePtr -> data )
          nextPtr = nodePtr -> leftChild // make the leftmost sub-tree the next sub-tree
        else
          nextPtr = nodePtr -> rightChild; // make the rightmost sub-tree the next sub-tree
      }
    /*------------------ insert the new value into the BST if it is not found in it --------*/
    if( !found )
    {
      TreeNode *newPtr = new TreeNode;
      newPtr -> data = item;
      newPtr -> leftChild = newPtr -> rightChild = NULL;
      if( item < nodePtr -> data )
        nodePtr -> leftChild = newPtr; // insert to the left of the parent node
      else
        nodePtr -> rightChild = newPtr; // insert to the right of the parent node
    }
  }
}
```
**Exercise B6**

1. Draw the BST that results when the following keywords are inserted in the order given:
   a. if, do, goto, case, switch, while, for.
   b. do, case, for, if, switch, while, goto.
   c. While, switch, goto, for, if, do, case.
2. Show each BST in question 1 with the keywords break, new, and public inserted into it.

**Removing a Node with a Given Value from a BST**

- In order to remove a node with a given value from a BST, you must first search the tree to find out if there is a node in the tree with that value.
- If you find such a node, you must also find out the address of its parent node.
- There are three types of nodes in a BST:
  1. Leaf nodes: the leftmost and the rightmost pointer variable are set to NULL.
  2. Nodes with only one child: the leftmost or rightmost pointer variable is set to NULL.
  3. Nodes with two children.
- We now provide the definition of the private member function that searches a BST for a given value and returns the result of the search (found/not found), and if it is found, the address of the node where it is found, and the address the parent node of this node. If the value is found at the root node, the address of the parent node is set to NULL.

```cpp
void searchNode( dataType item, bool &isfound, TreeNode * &locPtr, TreeNode * &parentPtr )
{
    locPtr = bstRoot;
    parentPtr = NULL;
    isfound = false;
    while( !isfound && locPtr ) // isfound == false && locPtr != NULL
    {
        if( item == locPtr->data )
            isfound = true;
        else  // move to the left/right sub-tree
        {
            parentPtr = locPtr;
            if( item < locPtr->data )
                locPtr = locPtr->leftChild;
            else
                locPtr = locPtr->rightChild;
        }
    }
}
```
Removing a Leaf Node

- You remove the leaf node with the address in the pointer variable `nodePtr` and the address of its parent node in the pointer `parentNodePtr` as follows:

```c
if( parentNodePtr == NULL ) // we are removing the root node: so the tree becomes empty
    bstRoot = NULL;
else if( parentNodePtr -> leftChild == nodePtr )
    parentNodePtr -> leftChild = NULL;
else
    parentNodePtr -> rightChild = NULL;
delete nodePtr;
```

Example

![BST on the left with the node with value 18 removed:](image)

Removing a Node with One Child

- You remove the node (with one child) with the address in the pointer variable `nodePtr` and the address of its parent node in the pointer `parentNodePtr` as follows:

```c
TreeNode *nextPtr; // to hold the address of the child node
/*--------------------------------------- first find out where the child is: left/right ---------------------------------------*/
if( nodePtr -> leftChild != NULL ) // it is a left child
    nextPtr = nodePtr -> leftChild;
else // it is a right child
    nextPtr = nodePtr -> rightChild;
if( parentNodePtr == NULL ) // the root node is removed: the child node becomes the root node
    bstRoot = nextPtr;
else if( parentNodePtr -> leftChild == nodePtr ) // the leftmost child is removed
    parentNodePtr -> leftChild = nextPtr;
else // the rightmost child is removed
    parentNodePtr -> rightChild = nextPtr;
delete nodePtr;
```
- Note that this code can also be used to remove a leaf node from a BST: because since the leftmost child and the rightmost child are NULL, the pointer variable `nextPtr` will be set to NULL.

### Example

![BST on the left with the node with value 14 removed:](image)

#### Removing a Node with Two Children

- You remove the node (with two children) with the address in the pointer variable `nodePtr` and the address of its parent node in the pointer `parentNodePtr` as follows:

  1. Search the BST to find the node that comes after this node (successor node) in the in-order traversal of the tree and its parent node.
  2. Replace the value of the node to be removed by the value of its successor node.
  3. Remove its successor node.

- Note that removing the successor node corresponds to the case of removing a node with one or no child; because its successor node cannot have a left child. Otherwise that left child would have been the successor node.

### Example

![BST on the left with the node with value 10 removed:](image)
The code segment follows:

```c
TreeNode  *succPtr ,  // hold the address of the successor node
     *succParentPtr;  // hold the address of the parent of the successor node

succPtr  =  nodePtr  -  rightChild;
succParentPtr  =  nodePtr;
while(  succPtr  -  leftChild  != NULL )
{
    succParentPtr  =  succPtr;
    succPtr  =  succPtr  -  leftChild;
}

/*----- Move the data part of the successor node to the node marked for deletion --------*/
/*----------------------------- and mark the successor node for deletion ----------------*/
nodePtr  -  data   =   succPtr  -  data;
nodePtr  =  succPtr;
parenNodePtr  =  succParentPtr;

/*---------- Proceed to remove a node with zero or one child ---------------------*/
TreeNode  *nextPtr;  // to hold the address of the child node

/*---------------- first find out where the child is: left/right ---------------------*/
if(  nodePtr  -  leftChild  !=  NULL )  // it is a left child
    nextPtr  =  nodePtr  -  leftChild;
else  // it is a right child
    nextPtr  =  nodePtr  -  rightChild;

if(  parenNodePtr  ==  NULL )  // the root node is removed: the child node becomes the root node
    bstRoot  =  nextPtr;
else  if(  parenNodePtr  -  leftChild  ==  nodePtr )  // the leftmost child is removed
    parenNodePtr  -  leftChild  =  nextPtr;
else  // the rightmost child is removed
    parenNodePtr  -  rightChild  =  nextPtr;

delete nodePtr;
```

the member function is defined as follows:
void removeNode( dataType value )
{
    TreeNode *nodePtr,
        *parentNodePtr;
    bool found;

    /*--------------------------------------
    find out if the value is in the BST
    --------------------------------------*/
    searchNode( value, found, nodePtr, parentNodePtr);

    if( !found ) // It is not in the BST
    {
        cerr << endl << "The value is not in the BST";
        exit( 1);
    }

    /*--------------------------
    Check if the node to be removed has two children
    --------------------------*/
    if( nodePtr -> leftChild && nodePtr -> rightChild )
    {
        /*---find the successor (in the LVR traversal) of the node to be removed and its parent node----*/
        TreeNode *succPtr, // hold the address of the successor node
            *succParentPtr; // hold the address of the parent of the successor node
        succPtr = nodePtr -> rightChild;
        succParentPtr = nodePtr;
        while( succPtr -> leftChild != NULL )
        {
            succParentPtr = succPtr;
            succPtr = succPtr -> leftChild;
        }
        /*--- Move the data part of the successor node to the node marked for deletion */
        /*---------------------------- and mark the successor node for deletion ----------------*/
        nodePtr -> data = succPtr -> data;
        nodePtr = succPtr;
        parentNodePtr = succParentPtr;
    }
TreeNode *nextPtr; // to hold the address of the child node

/*----------------------------------*/
Proceed to remove a node with zero or one child ----------------------------------*/

/*----------------------------*/
first find out where the child is: left/right ----------------------------------*/

if( nodePtr -> leftChild != NULL ) // it is a left child
  nextPtr = nodePtr -> leftChild;
else // it is a right child
  nextPtr = nodePtr -> rightChild;

if( parentNodePtr == NULL ) // the root node is removed: the child node becomes the root node
  bstRoot = nextPtr;
else if( parentNodePtr -> leftChild == nodePtr ) // the leftmost child is removed
  parentNodePtr -> leftChild = nextPtr;
else // the rightmost child is removed
  parentNodePtr -> rightChild = nextPtr;

delete nodePtr;

} 

Exercise B7

Given the following binary search tree:

```
          10
         / \
        /   \
       8     20
      / \
     /   \
    7    9   14
   / \
  /   \
5     12 18
   / \
  /   \
6    11 17
   \
8
```

Show the tree after each of the following values is removed from it: 11, 7, 8, 10, and 14.
Exercise B8

Answer the following questions given that the nodes of a BST are defined using the following structure:

```c
typedef int   dataType;

struct TreeNode
{
    dataType data;   //data part
    TreeNode *leftChild;   // left child address
    TreeNode *rightChild;  // right child address
};
```

1. Write the sequence of statement(s) to create an empty BST.
2. Write the function `void inorderTraversal( ostream & out, TreeNode * subtreeRoot )` that outputs the values of the nodes of a BST by traversing the tree in in-order traversal.
3. Write the function `void preorderTraversal( ostream & out, TreeNode * subtreeRoot )` that outputs the values of the nodes of a BST by traversing the tree in pre-order traversal.
4. Write the function `void postorderTraversal( ostream & out, TreeNode * subtreeRoot )` that outputs the values of the nodes of a BST by traversing the tree in post-order traversal.
5. Write the function `bool searchBST( dataType item, TreeNode * root )` that returns true if the value received in parameter `item` is in the BST with the address of the root node received in the parameter `root`, and false otherwise.
6. Write the function `void insertNodeAfterParent( dataType value, TreeNode * parentNode) that receives a value and inserts a node with that value as a child node of the node with address received in the parameter `parentNode`.
7. Write the function `void remove1( TreeNode * nodePtr, TreeNode *parentNodePtr)` that removes the leaf node with address in the pointer variable `nodePtr` and the address of its parent node in the pointer `parentNodePtr`.
8. Write the function `void remove2( TreeNode * nodePtr, TreeNode *parentNodePtr)` that removes the node (with one child node) with address in the pointer variable `nodePtr` and the address of its parent node in the pointer `parentNodePtr`.
9. Write the function `void remove3( TreeNode * nodePtr, TreeNode *sucPtr, TreeNode *sucparentPtr)` that removes the node (with two children nodes) with address in the parameter `nodePtr` provided that the address of its successor (in the LVR traversal of the tree) is in the parameter `sucPtr` and the address of the parent node of this successor is in the parameter `sucparentPtr`.
Hash Tables

- A **Hash table** is an array to hold data such that the location of each data item in the array is determined by a function on data items (or some key fields on data items) called **hash function**.

**Example**

The department of computer science at a university wants to keep track of CS majors at the university by using a record with the following information:
- A student’s ID number (a four-digit number)
- His last name
- His first name
- His GPA

This information is represented by the following structure:

```cpp
struct StudentInfo {
    int idNum;
    string lastName;
    string firstName;
    double gpa;
};
```

To hold the information about CS majors, an array of `StudentInfo` structures is created as follows:

```cpp
StudentInfo studentRecords[10000]; // hash table
```

And the information about a student is entered in the table at index his ID number. That means that the hash function `h` is defined on a `StudentInfo` structure variable `student` as follows:

```cpp
h(student) = student.idNum
```

- The **purpose of hashing** is to store data in such a way that the time required to search the table for a given value (or record) is constant (O(1)): That means that it does not depend on the size of the data.
- But in general, time efficiency is usually achieved at the detriment of space efficiency:
  - In the example above, if there are 250 CS majors then we are using only 250 elements in an array of 10000 elements: only 2.5% of the array.
- A tradeoff is therefore needed between time and space efficiencies.
- One tradeoff approach is to reduce the size of the hash table and to allow two or more data items to be assigned to the same location in the array (a process known as **collision**).
For the example above, if we use the hash table

\[
\text{StudentInfo } \ast\text{studentListHeads[ 250 ];} \quad \text{// hash table}
\]

And the hash function: \( h(\text{student}) = \text{student.idNum} \mod 250 \)

There will be no waste of space; but two or more student records can be assigned to the same location in the array: for example, IDs 0000, 0250, 0500 are all assigned to index 0 in the array.

- Two strategies are used to deal with collisions: the open-addressing strategy and chaining.

- You use chaining to deal with collisions as follows:
  - Create a table of linked lists as the hash table.
  - Each data item is appended to the linked list that corresponds to hash function location.

For the example above, if we use the hash table

\[
\text{StudentInfo } \ast\text{studentListHeads[ 250 ];} \quad \text{// hash table}
\]

And the hash function: \( h(\text{student}) = \text{student.idNum} \mod 250 \)

Then we may have the following linked lists:

![Linked list diagram](image-url)
With chaining as the collision strategy, to search for a data item in the hash table as follows:

- You first use the hash function to compute the location of the head of the linked list where it might be located.
- Then you search that linked list to find out if it is there.

**Hash Functions**

- Another important factor in the design of a hash table is the selection of the hash function.
- An ideal hash function is a function that is easy to evaluate and that generates few collisions.
- For strings, a hash function is in general designed by using the ASCII codes of the characters in a string or by assigning the code 1 to ‘A’, 2 to ‘B’, . . . , and 26 to ‘Z’.

The following are examples of hash function on strings:

\[
h_1(\text{name}) = \frac{(\text{first-letter} + \text{last-letter})}{2} \mod (\text{size of the table})
\]

\[
h_2(\text{name}) = \frac{\text{sum of all letters}}{\text{length}} \mod (\text{size of the table})
\]

**Example**

Using a table of 7 elements, we have the following:

\[
h_1(\text{"JOHN"}) = \frac{(10 + 14)}{2} \mod 7 = 5
\]

\[
h_2(\text{"JOHN"}) = \frac{(10 + 15 + 8 + 14)}{4} \mod 7 = 4
\]

\[
h_1(\text{"P"}) = h_2(\text{"P"}) = \frac{16}{7} = 2
\]

**Exercise H1**

- Using a hash table with 7 locations and the hash function \( h(i) = i \mod 7 \), show the hash table that results when the integers are inserted in the order given, with collisions resolved using chaining: 5, 11, 18, 23, 28, 13, 25.

- Using a hash table with 11 locations and the hash function \( h(i) = i \mod 11 \), show the hash table that results when the integers are inserted in the order given, with collisions resolved using chaining: 26, 42, 5, 44, 92, 59, 40, 36, 12, 60, 80.

- Assuming that codes are assigned to characters as follows, 1 to ‘A’, 2 to ‘B’, . . . , and 26 to ‘Z’ and that the hash table has 11 elements and the hash function is:

\[
h(\text{name}) = \frac{(\text{first-letter} + \text{last-letter})}{2} \mod 11
\]

Show the hash table that results when the identifiers are inserted in the order given, with collisions resolved using chaining: BETA, RATE, FREQ, ALPHA, MEAN, SUM, NUM, BAR, WAGE, PAY, KAPPA.
Problem of Lopsidedness with BST

- The order in which items are inserted into a BST determines the shape of the tree.
- For example, insert the following characters into a BST of characters in the given order:
  - O, E, T, C, U, M, and P
  - C, O, M, P, U, T, and E
  - C, E, M, O, P, T, and U.
- The time required to carry out most of the basic operations on a BST depends on the shape of the tree:
  - If the tree is balanced so that the left and right sub-trees of each node contain approximately the same number of nodes, then the computing time of each of the operations search( ), insert( ), and delete( ) is $O(\log_2 n)$.
  - On the other hand, if the tree is represented as a linked list, then their computing times will be $O(n)$.
- The usual solution is to rebalance the tree after each new element is inserted using some rebalancing algorithm.
- One such algorithm is due to two Russian mathematicians, Georgii Maksimovich Adel’son-Vel’skii and Evgenii Mkhailovich Landis and the resulting tree is named AVL (or height-balanced) tree.
- Given a BST, the balance factor of a node $N$ is the height of the left subtree of $N$ minus the height of its right subtree.
- An AVL (or height balanced) tree is a BST tree such that the balance factor of each node is -1, 0, or 1.
- The following are examples of AVL trees:
The following are not AVL trees:

The Basic Rebalancing Rotations

- When insertion of a new item causes an imbalance in a binary tree, the tree can be rebalanced by applying a rotation to the subtree whose root has a balance factor of +/-2 and is the nearest ancestor of the inserted node.

- The four type of rotations are:
  1. Simple right rotation: use when the inserted item is in the left subtree of the left child of the nearest ancestor with a balancing factor of +2.

  2. Simple left rotation: use when the inserted item is in the right subtree of the right child of the nearest ancestor with a balancing factor of -2.
3. **Left-right rotation**: use when the inserted item is in the right subtree of the left child of the nearest ancestor with a balancing factor of +2.

4. **Right-left rotation**: use when the inserted item is in the left subtree of the right child of the nearest ancestor with a balancing factor of -2.
Rebalancing Rules (by Gilbert Ndjatou)

1. If the balance factor of a node is +2, do the following:
   a. Create a new node with the value of this node
   b. Make this new node the right child of the current node
   c. Replace the value of the current node with the largest value in its left-subtree.
   d. Remove the node with the largest value in its left-subtree

2. If the balance factor of a node is -2, do the following:
   e. Create a new node with the value of this node
   f. Make this new node the left child of the current node
   g. Replace the value of the current node with the smallest value in its right-subtree.
   h. Remove the node with the smallest value in its right-subtree

Application

Insert the following state abbreviations into an AVL tree: RI, PA, DE, GA, OH, MA, IL, MI, IN, and NY.

Exercises

Trace the construction of the AVL tree that results from the insertion of the following collection of numbers in the given order. Show the tree and balance factors for each node before and after rebalancing.

a. 22, 44, 88, 66, 55, 11, 99, 77, 33
b. 11, 22, 33, 44, 55, 66, 77, 88, 99.
c. 99, 88, 77, 66, 55, 44, 33, 22, 11.
d. 55, 33, 77, 22, 11, 44, 88, 66, 99
e. 50, 45, 75, 65, 70, 35, 25, 15, 60, 20, 41, 30, 55, 10, 80.
Directed Graphs

- A directed graph or digraph consists of a finite set of elements called vertices, or node, together with a finite set of directed arcs, or edges that connect pairs of vertices.

Example

Note that trees are special kinds of directed graphs.
Trees are characterized by the fact that one of their nodes (the root) has no incoming arc and every other node can be reached from the root by a unique path (sequence of edges) from the root.

The following are basic operations on a digraph:
- Digraph traversal: visiting each node exactly once.
- Determining whether one node is reachable from another node.
- Determining the number of paths from one node to another.
- Finding a shortest path from one node to another.

Digraphs are often represented using adjacency-Matrices or adjacency-lists.

Adjacency-Matrix Representation of a Digraph

- You construct the adjacency-matrix of a digraph as follows:
  1. First number the vertices of the graph 1, 2, 3, . . . , n.
  2. The adjacency-matrix is the n x n matrix adj in which the entry in row i and column j is 1 (or true) if vertex j is adjacent to vertex i (that means that there is a directed arc from vertex i to vertex j).

Example
The adjacency-matrix of the graph above is:
For a weighted digraph in which some “cost” or “weight” is associated with each arc, the cost of the arc from vertex i to vertex j is used instead of 1 in the adjacency matrix.

The matrix representation of a graph is useful to solve a variety of graph problems:

- It is easy to determine the in-degree of a vertex (the number of edges coming into that vertex) and the out-degree of a vertex (the number of edges meaning from that vertex).
- Path counting problem (the number of paths of a certain length from one node to another): the number of paths of length \( n \) (\( n \geq 1 \)) from node \( i \) to a node \( j \) is the entry in row \( i \) and column \( j \) in the \( n \)th power of matrix \( \text{adj} \).

\[
\begin{array}{cccc}
0 & 1 & 0 & 1 \\
0 & 0 & 1 & 1 \\
0 & 0 & 0 & 0 \\
0 & 1 & 1 & 0 \\
0 & 1 & 1 & 0 \\
0 & 0 & 0 & 0 \\
0 & 1 & 1 & 0 \\
0 & 0 & 1 & 0 \\
0 & 1 & 1 & 0 \\
\end{array}
\times
\begin{array}{cccc}
0 & 1 & 0 & 1 \\
0 & 0 & 1 & 1 \\
0 & 0 & 0 & 0 \\
0 & 1 & 1 & 0 \\
0 & 1 & 1 & 0 \\
0 & 0 & 0 & 0 \\
0 & 1 & 1 & 0 \\
0 & 0 & 1 & 0 \\
0 & 1 & 1 & 0 \\
\end{array}
=
\begin{array}{cccc}
0 & 1 & 2 & 1 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 1 & 0 \\
\end{array}
\]

Searching and Traversing Digraphs

One standard problem with digraphs is to determine which nodes in a digraph are reachable from a given node. Two standard algorithms for searching for such vertices are \textbf{depth-first search} and breadth-first search.

**Depth-first Search Algorithm**

1. Visit the start vertex \( v \)
2. For each vertex \( w \) adjacent to \( v \), do the following:
   a. If \( w \) has not been visited, apply the depth- first search algorithm with \( w \) as the start vertex.
**Breadth-first Search Algorithm**

1. Visit the start vertex
2. Initialize a queue to contain only the start vertex
3. While the queue is not empty, do the following:
   a. Remove a vertex $v$ from the queue
   b. For all vertices $w$ adjacent to $v$ do the following:
      i. If $w$ has not been visited, then:
         ii. Visit $w$
         ii. Add $w$ to the queue

**Shortest-path Algorithm**

1. Visit the start vertex and label it 0.
2. Initialize the distance to 0
3. Initialize a queue to contain only the start vertex
4. While destination has not been visited and the queue is not empty, do the following:
   a. Remove a vertex $v$ from the queue
   b. If the label of $v$ is greater than distance, increment distance by 1
   c. For each vertex $w$ adjacent to $v$ do the following:
      i. If $w$ has not been visited, then:
         ii. Visit $w$ and label it with $distance + 1$
         ii. Add $w$ to the queue
5. If destination has not been visited then display “destination is not reachable from start vertex”.
   Else find the vertex $p[0], \ldots, p[distance]$ on the shortest path as follows:
   a. Initialize $p[distance]$ to destination
   b. For each value $k$, ranging from $distance - 1$ to 0
      Find vertex $p[k]$ adjacent to $p[k+1]$ with label $k$. 