04-630
Data Structures and Algorithms for Engineers

Lecture 11: Binary Search Trees

Adopted and Adapted from Material by:

David Vernon: www.vernon.eu

Lecture 10

Trees

- Types of trees
- Binary Tree ADT
- Recap: tree traversal
- Binary Search Tree
- Optimal Code Trees
- Huffman's Algorithm
- Height Balanced Trees
  - AVL Trees
  - Red-Black Trees

Recap: Tree Traversal

Tree Traversals

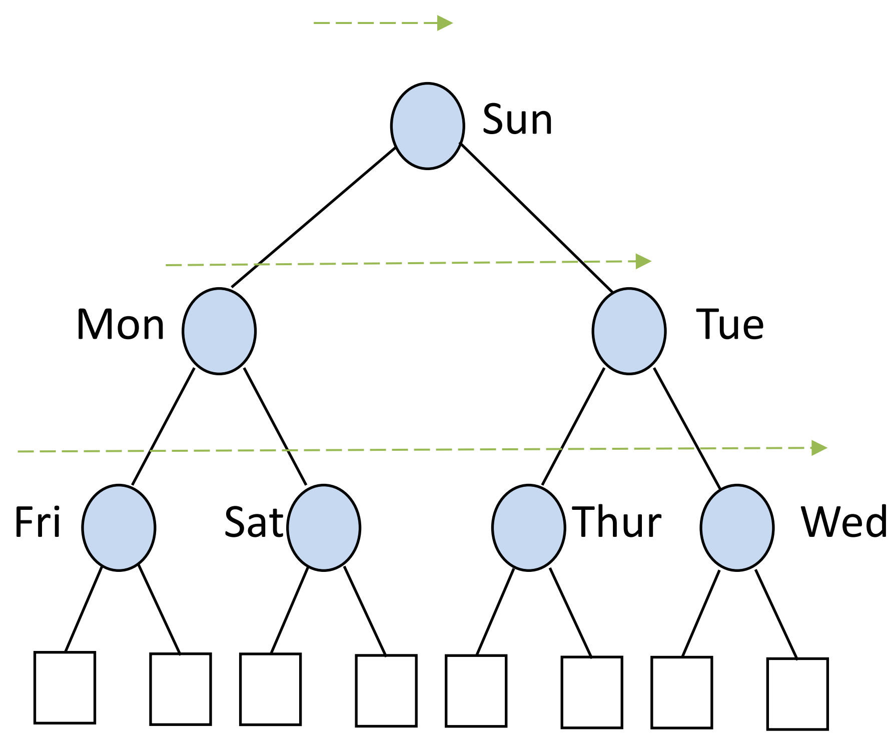
- To perform a traversal of a data structure, we use a method of visiting every node in some predetermined order

- Traversals can be used

  – to test data structures for equality
  – to display a data structure
  – to construct a data structure of a given size
  – to copy a data structure

Breadth-First traversal

- The traversal happens one level at a time.
- You traverse all children **at one level** before proceeding **to the grandchildren at the next level**.



BFS Traversal: [Sun, Mon, Tue, Fri, Sat, Thur, Wed]

Depth-First Traversals

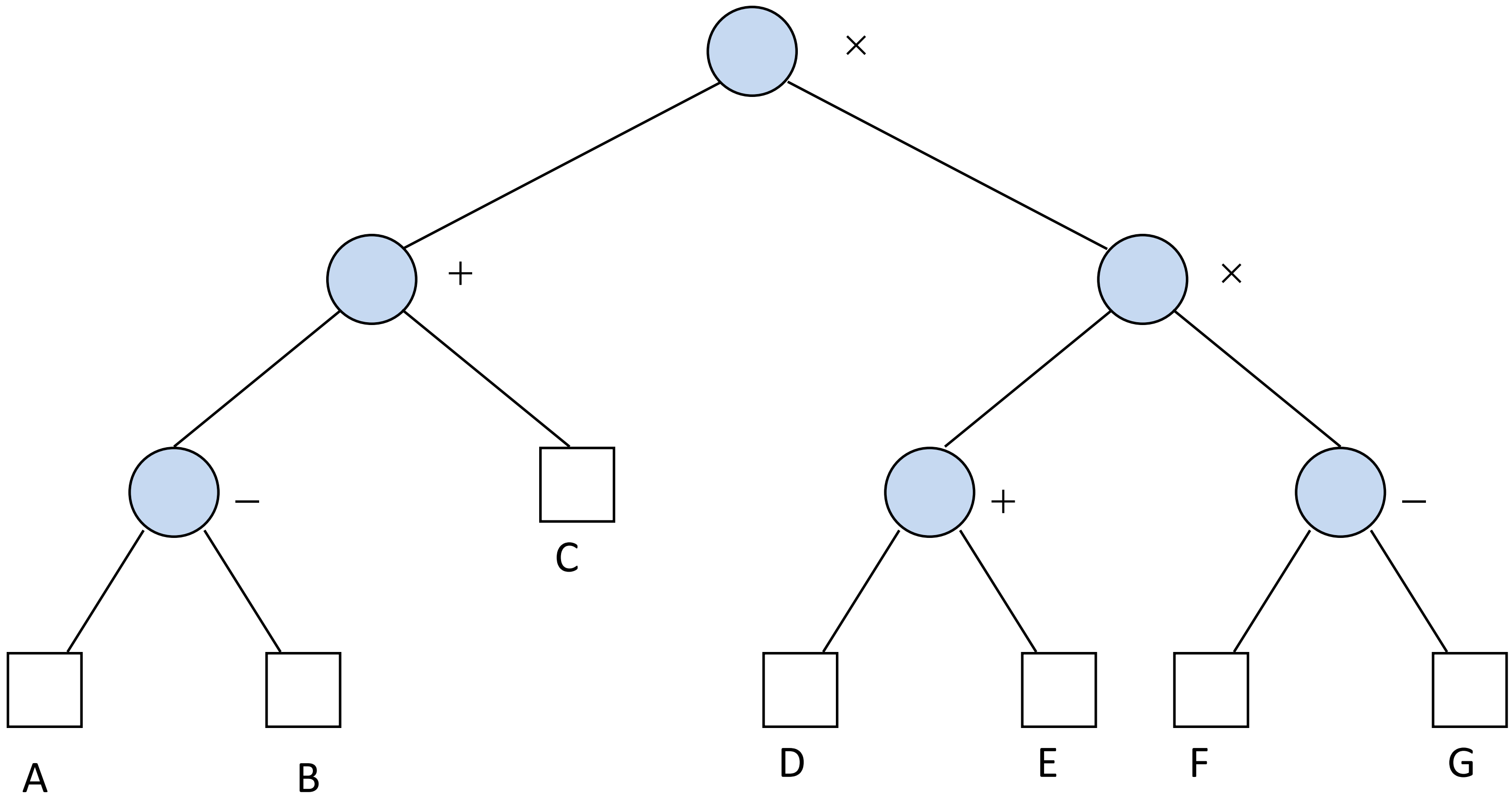
- Consider a binary tree.

- There are 3 depth-first traversals

  - **Pre-order traversal**: root then children(left, right)
  - **Post-order traversal**: children(left, right) then root.
  - **In-order traversal**: left child, root, right child

- For example, consider the expression tree:

Example: Expression Tree



Depth-First Traversals

- Inorder traversal

  A – B + C x D + E x F – G

- Postorder traversal

  A B – C + D E + F G – x x

- Preorder traversal

  x + –A B C x + D E – F G

Depth-First Traversals

- The parenthesised Inorder traversal

  ((A – B) + C) x ((D + E) x (F – G))

  This is the infix expression corresponding to the expression tree

- Postorder traversal gives a postfix expression

- Preorder traversal gives a prefix expression

Depth-First Traversals

Recursive definition of inorder traversal

Given a binary tree T

if T is empty

visit the external node

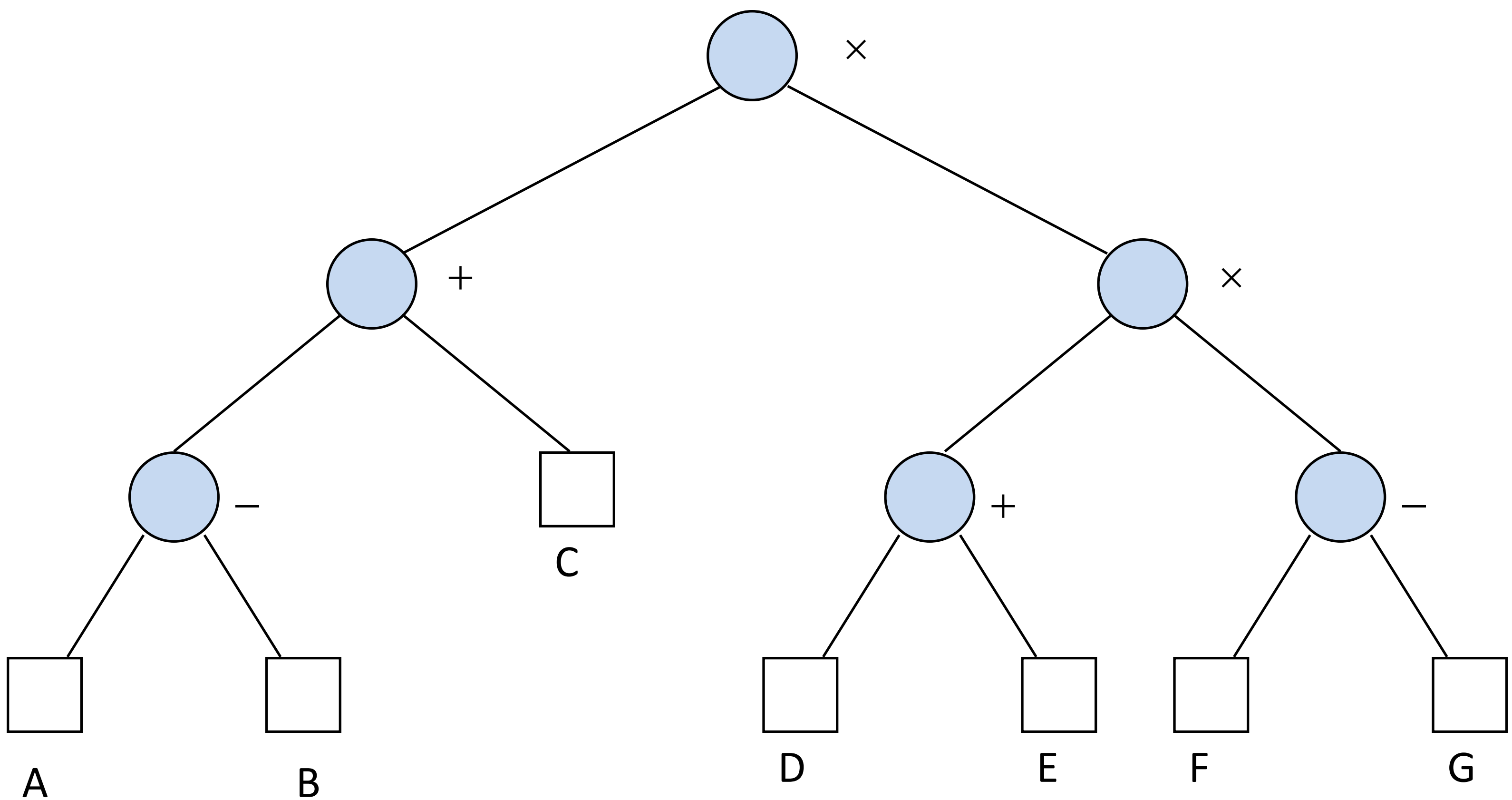
otherwise

perform an inorder traversal of $Left(T)$

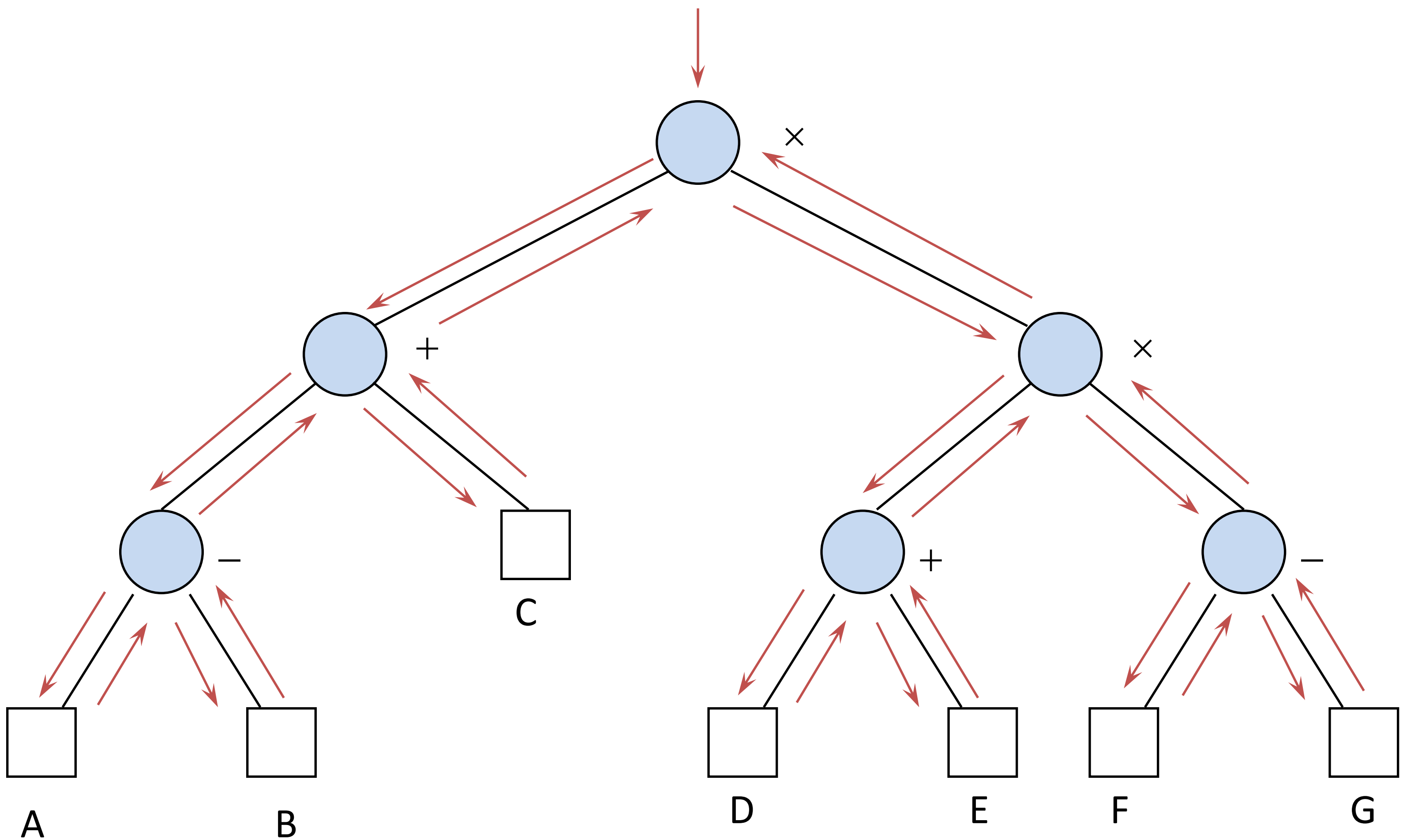
visit the root of T

perform an inorder traversal of $Right(T)$

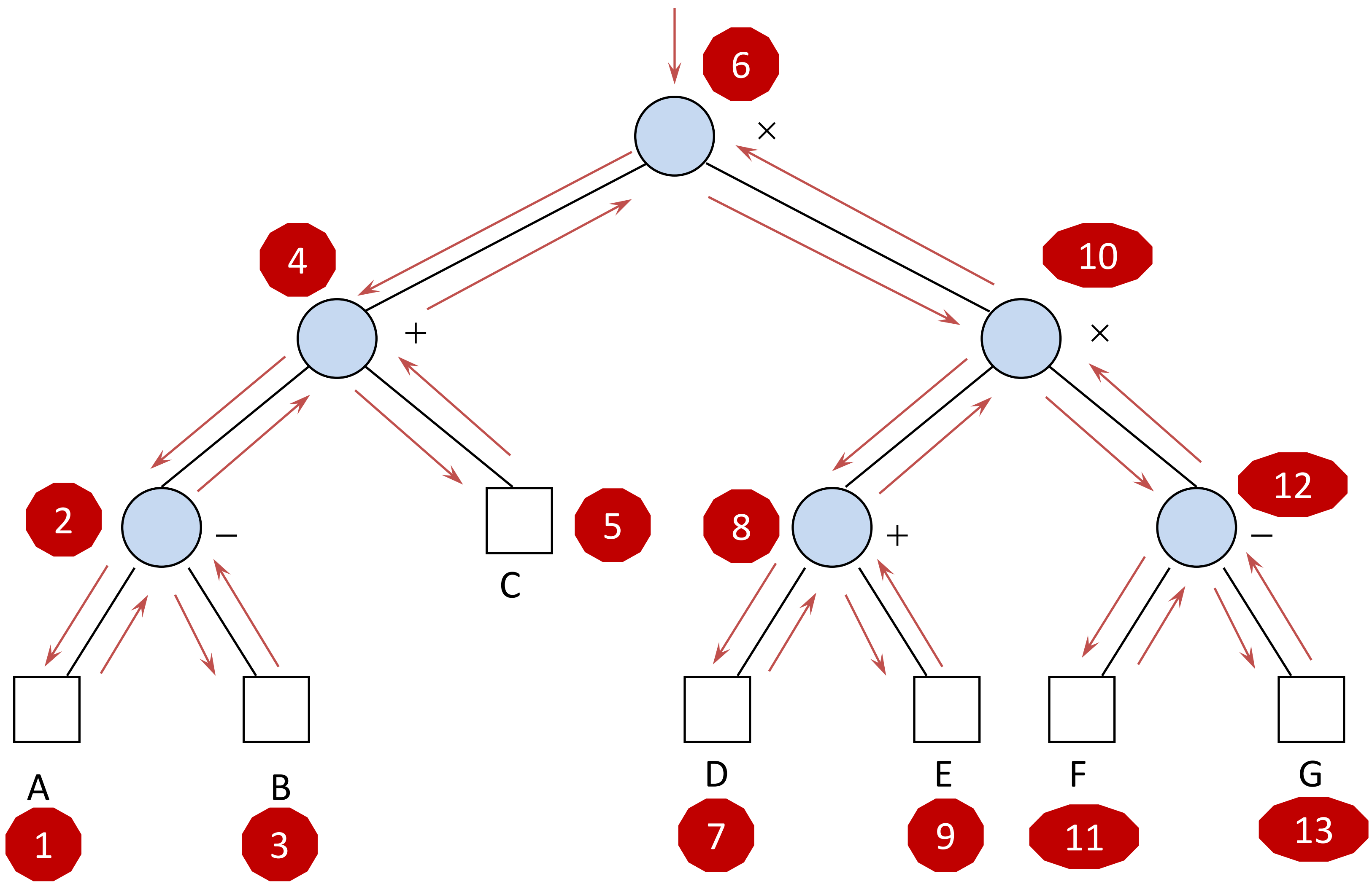
Example: Inorder Traversal



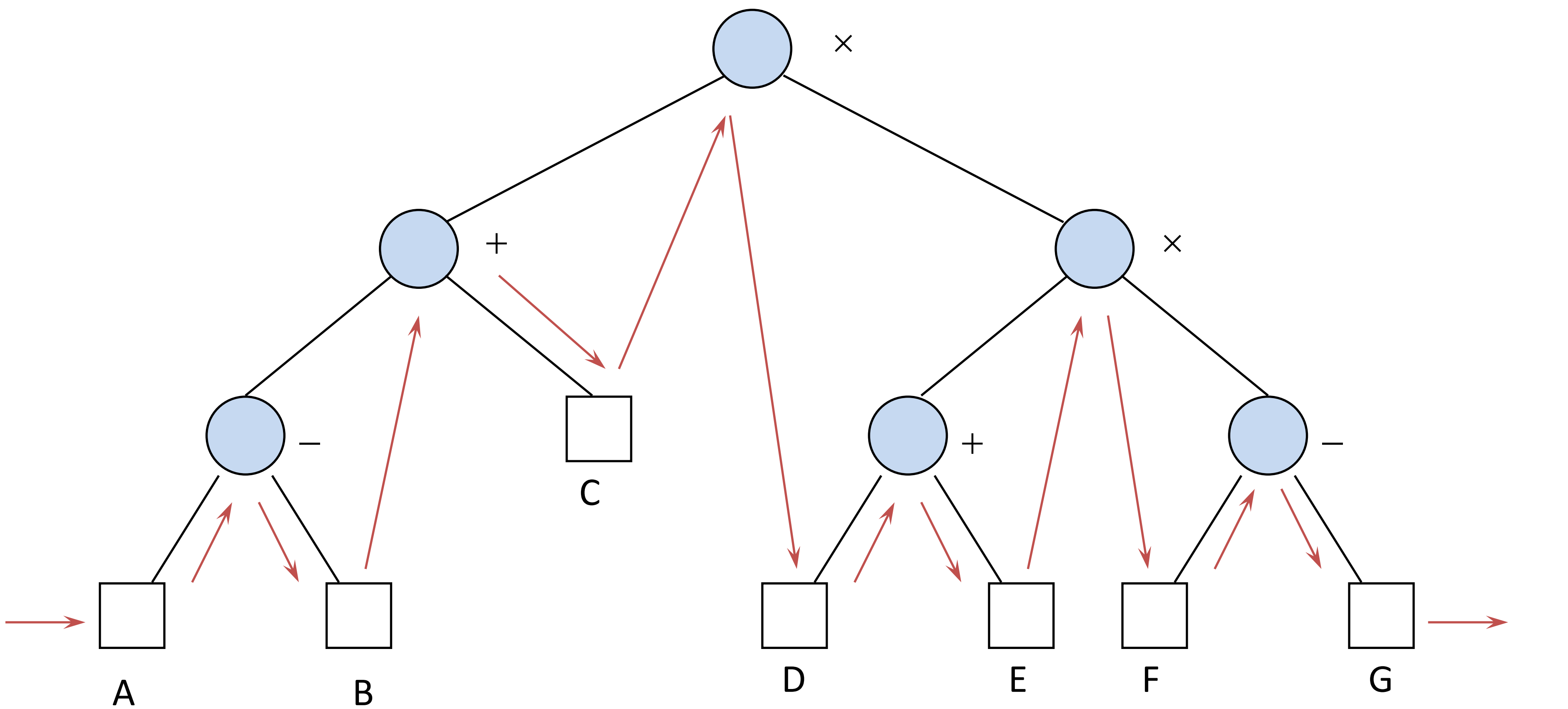
Example: Inorder Traversal



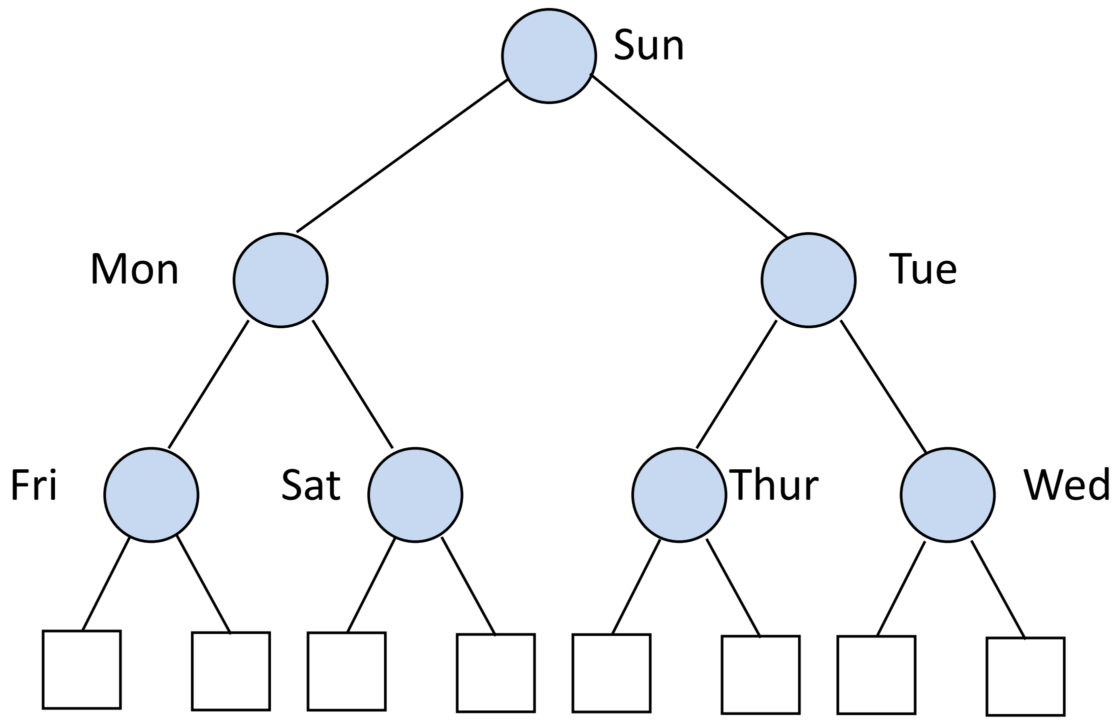
Example: Inorder Traversal



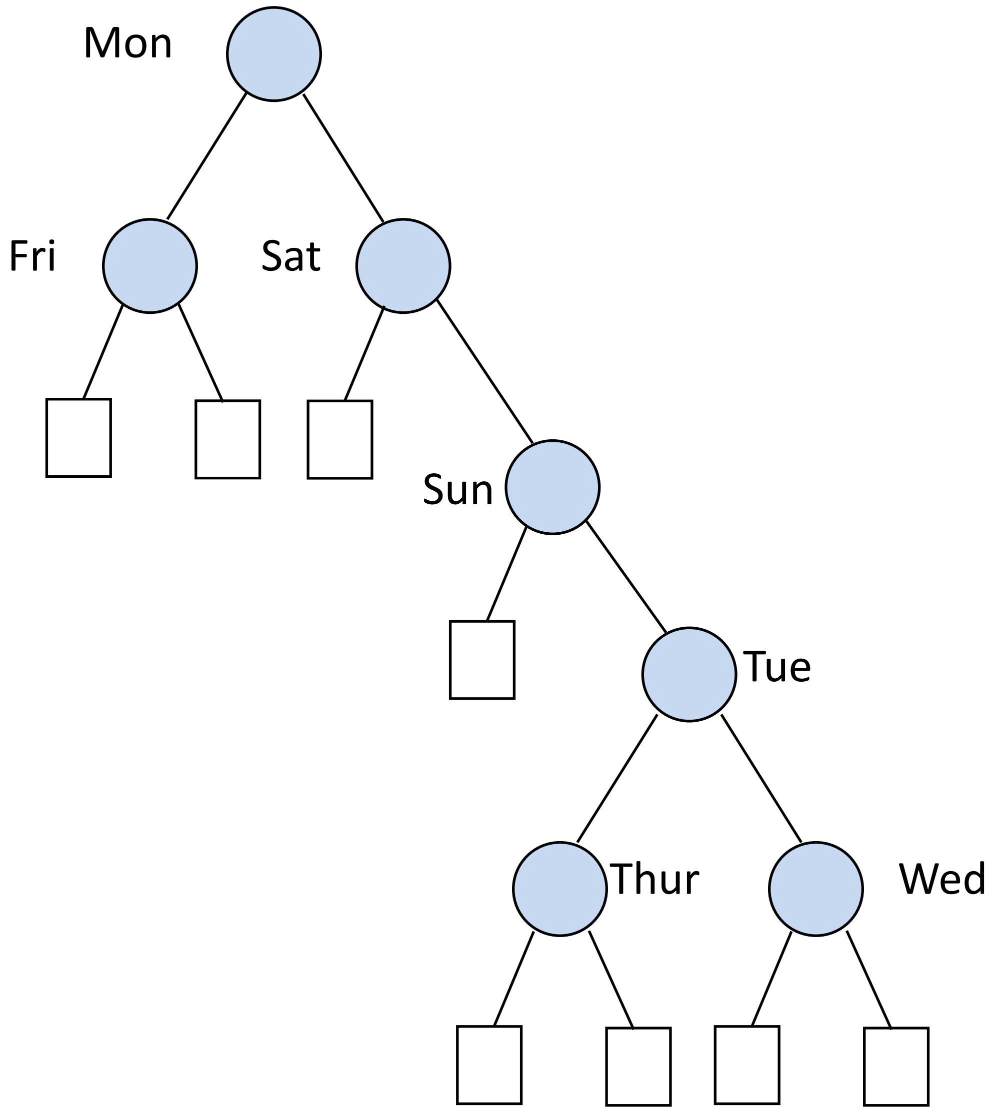
Example: Inorder Traversal



Example: Inorder Traversal



Example: Inorder Traversal



Depth-First Traversals

- Recursive definition of postorder traversal

  Given a binary tree T

  if T is empty

  visit the external node

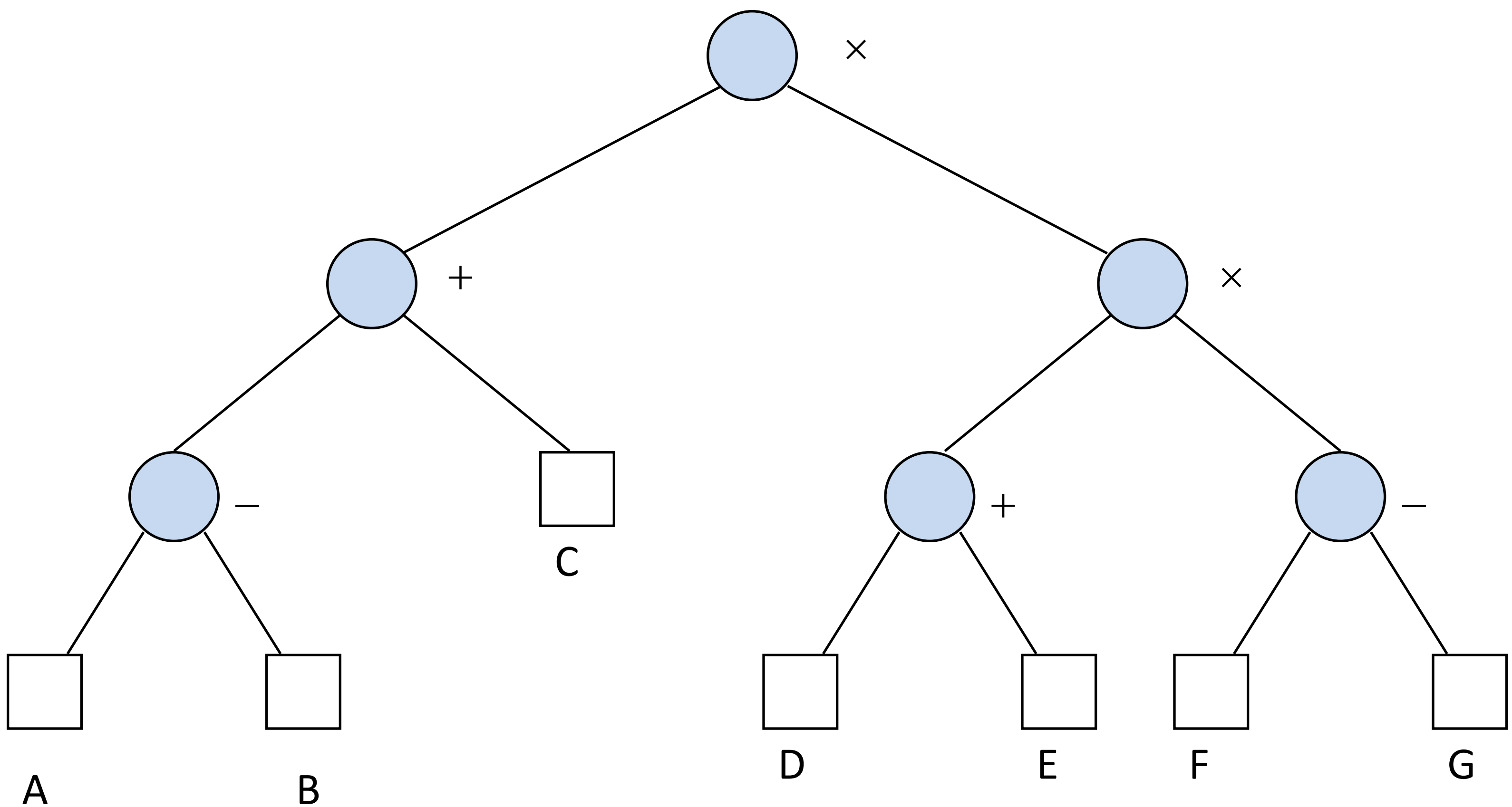
  otherwise

  perform an postorder traversal of $Left(T)$

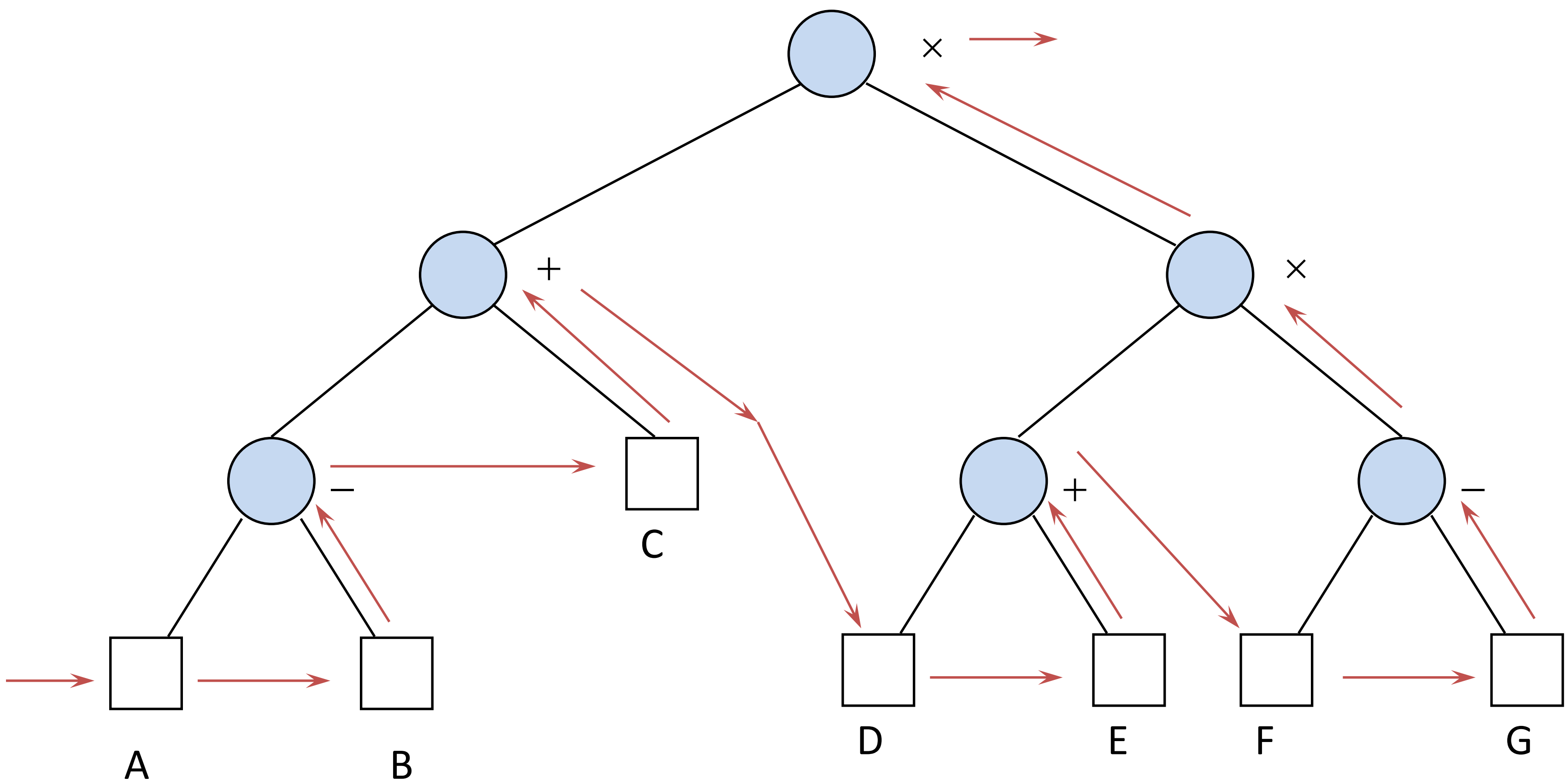
  perform an postorder traversal of $Right(T)$

  visit the root of T

Example: Postorder Traversal



Example: Postorder Traversal



Depth-First Traversals

- Recursive definition of preorder traversal

  Given a binary tree T

  if T is empty

  visit the external node

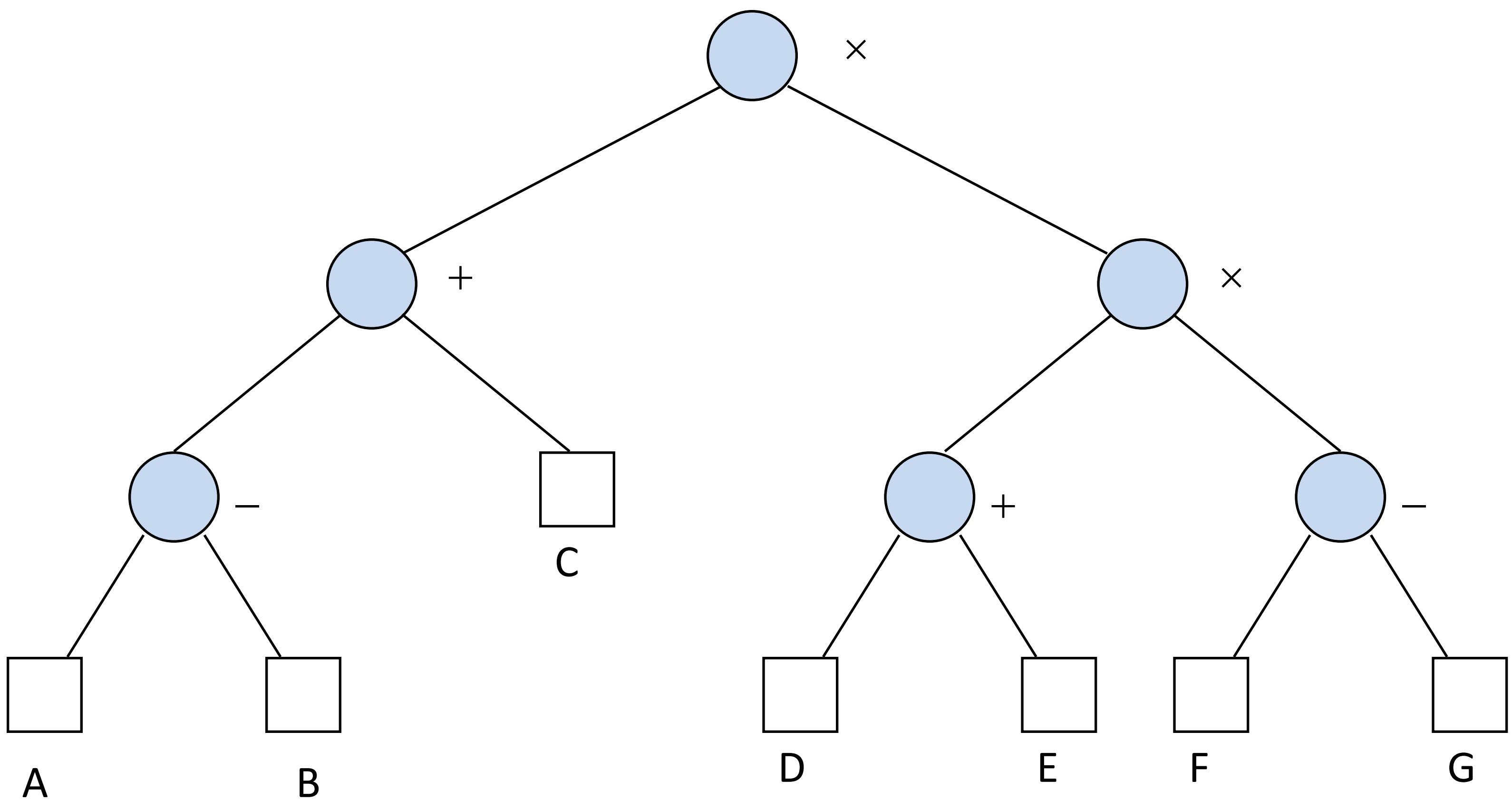
  otherwise

  visit the root of T

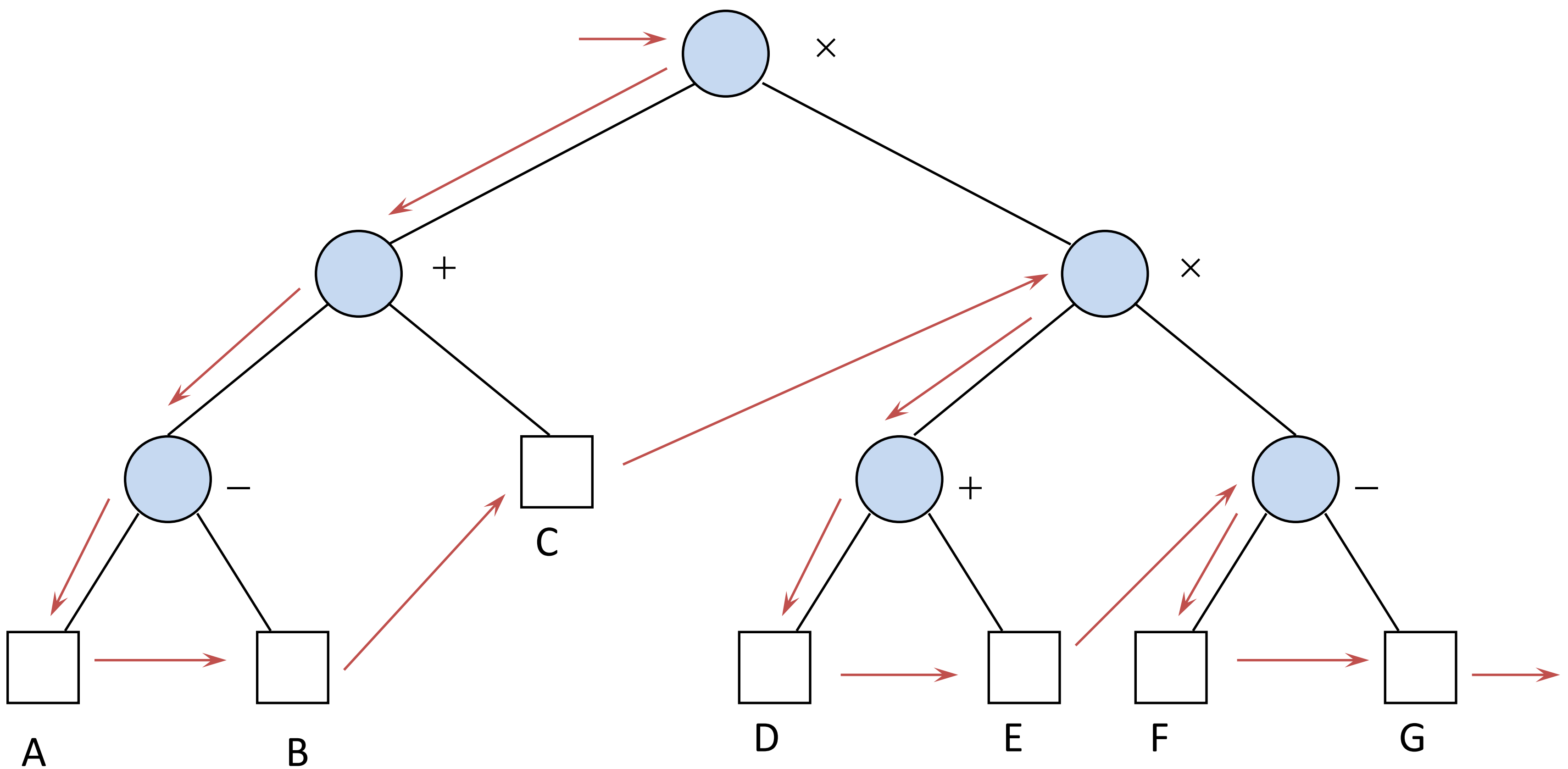
  perform an preorder traversal of $Left(T)$

  perform an preorder traversal of $Right(T)$

Example: Preorder Traversal

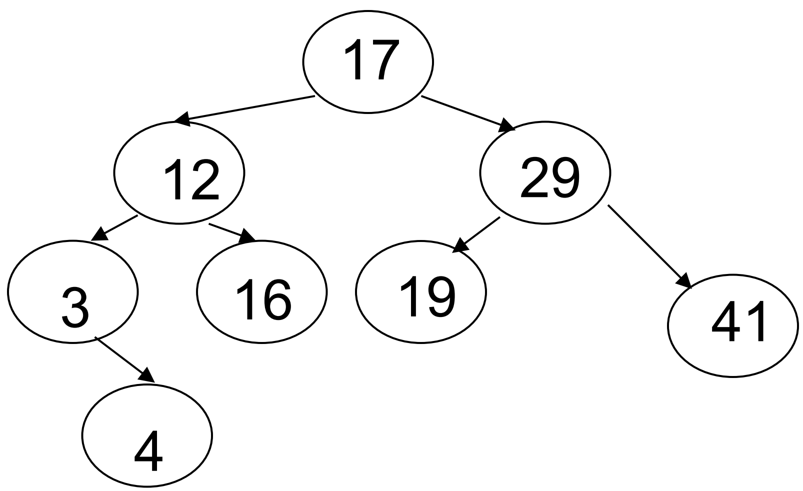


Example: Preorder Traversal



Exercise

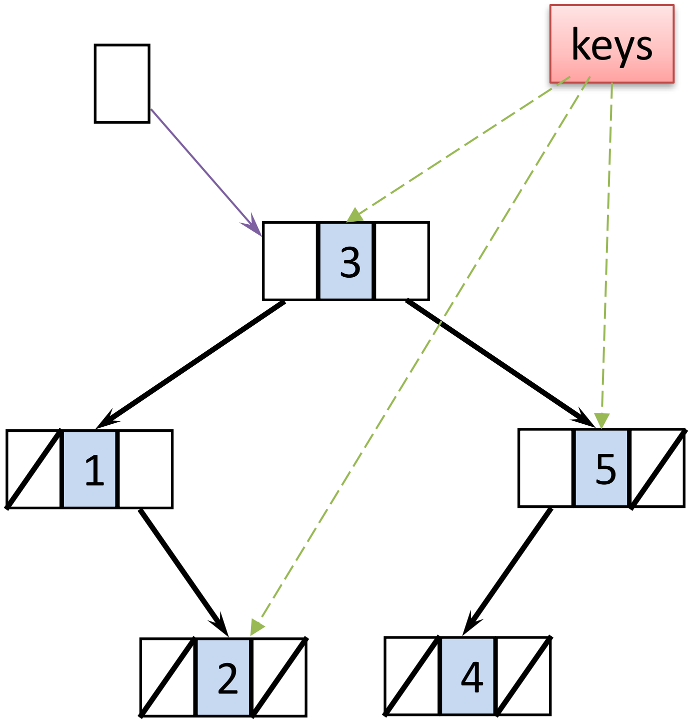
- Show the output of traversal using in-order, pre-order, and post-order traversal.



Binary Search Tree

Binary Search Trees

- A Binary Search Tree (BST) is a special type of binary tree
  - it represents information in an ordered format
  - A binary tree is a binary search tree if for every node w,
    - all keys in the left subtree of w have values less than the key of w
    - all keys in the right subtree have values greater than key of w.

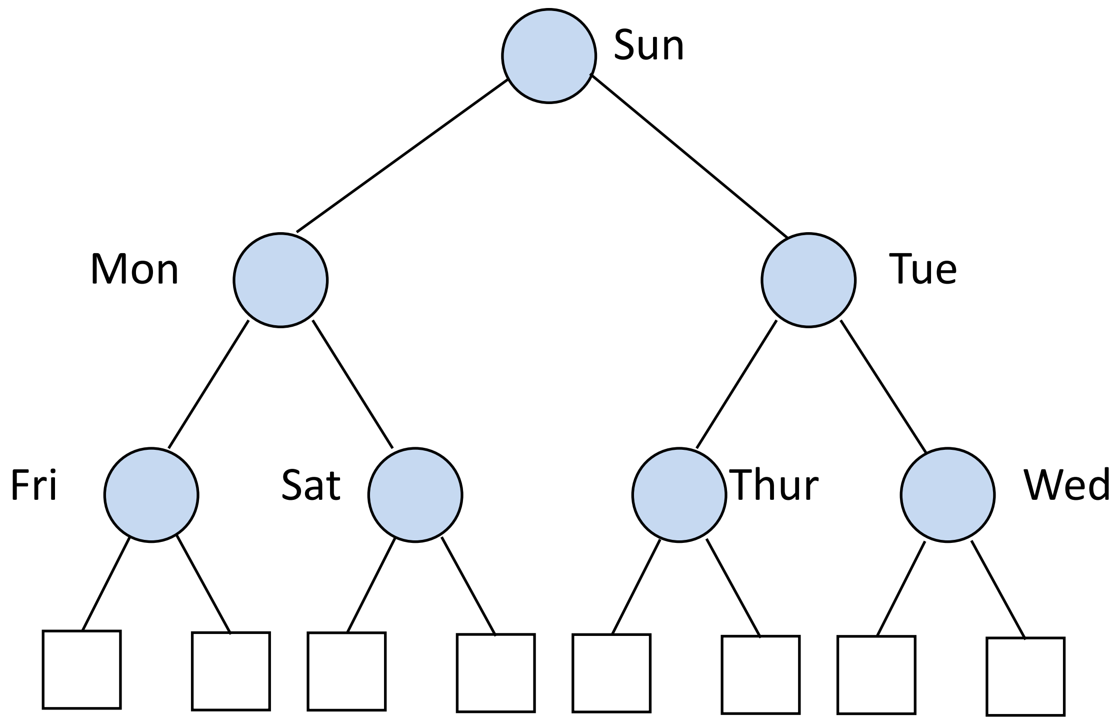


Binary Search Trees

Definition: A binary search tree T is a binary tree; either it is empty or each node in the tree contains an identifier and:

- all keys in the left subtree of T are less (numerically or alphabetically) than the identifier in the root node T;
- all identifiers in the right subtree of T are greater than the identifier in the root node T;
- The left and right subtrees of T are also binary search trees.

Binary Search Trees



Binary Search Trees

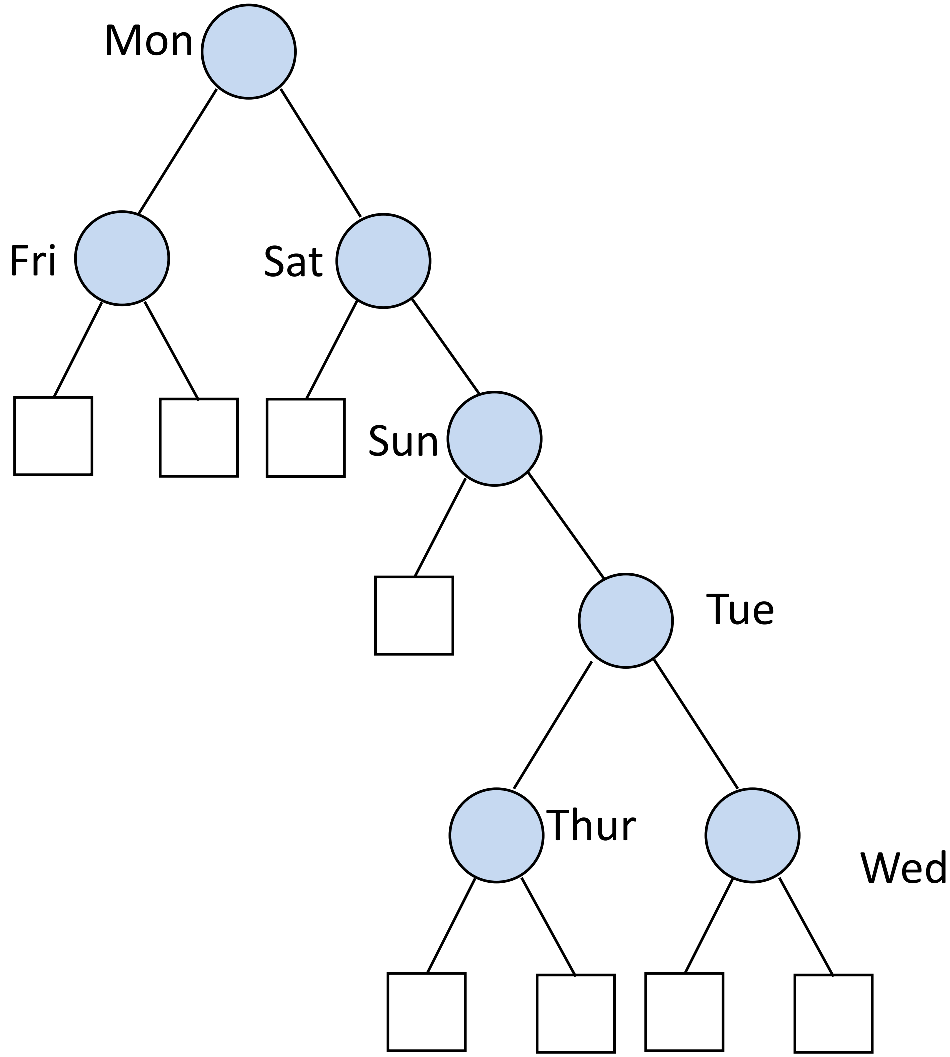
- The main point to notice about such a tree is that, if traversed inorder, the keys of the tree (*i.e.,* its data elements) will be encountered in a sorted fashion

- Furthermore, efficient searching is possible using the *binary search technique*
  - search time is $O(\log_2 n)$

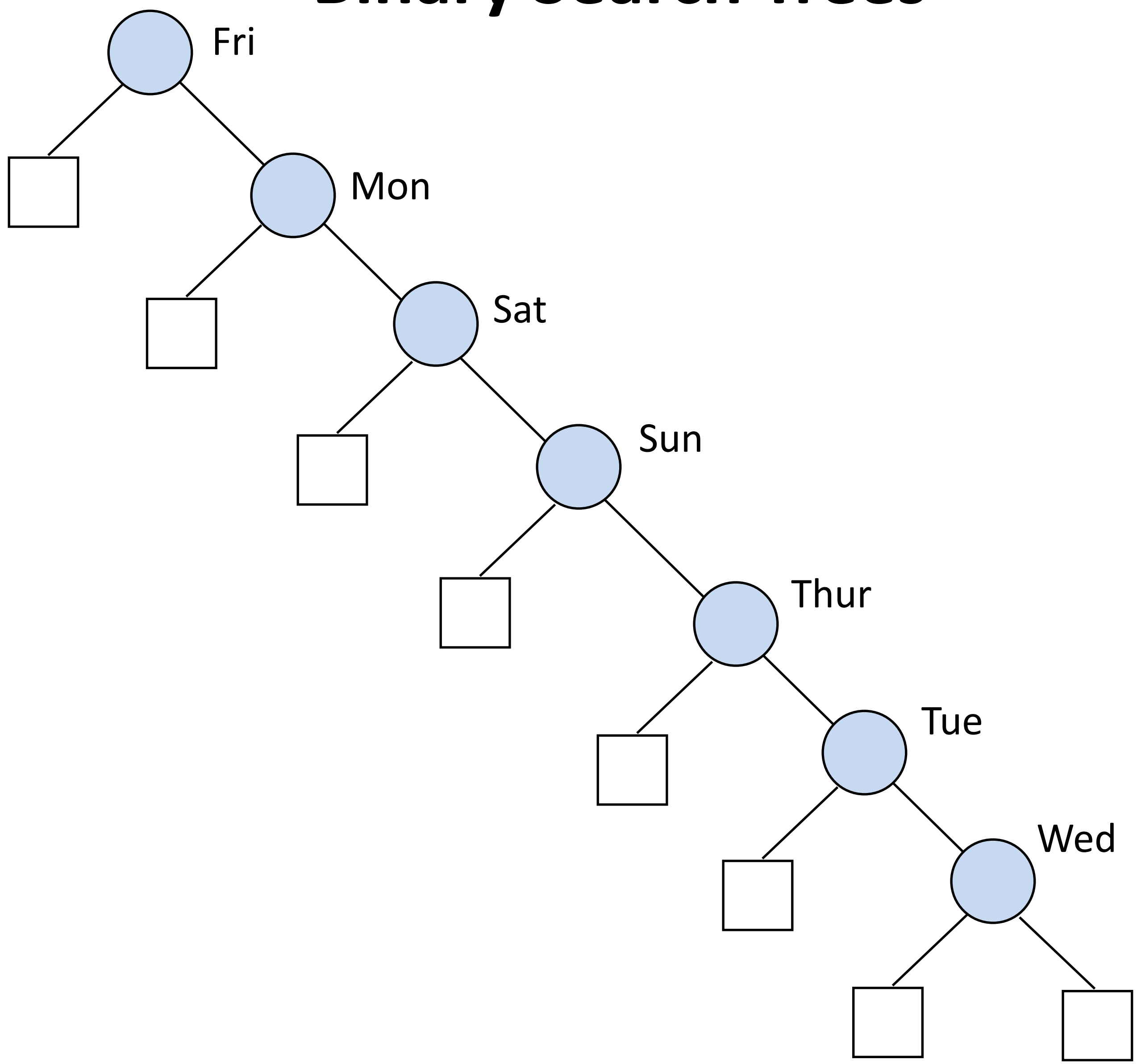
Binary Search Trees

It should be noted that several binary search trees are possible for a given data set, *e.g.,* consider the following tree:

Binary Search Trees



Binary Search Trees



Binary Search Trees

Let us consider how such a situation might arise

Construct a binary search tree:

- Assume we are building a binary search tree of words
- Initially, the tree is null, i.e., there are no nodes in the tree
- The first word is inserted as a node in the tree as the root, with no children

Binary Search Trees

On insertion of the second word, we check to see if it is the same as the key in the root, less than it, or greater than it

- If it is the same, no further action is required (duplicates are not allowed)
- If it is less than the key in the current node, move to the left subtree and *compare again*
- If the left subtree does not exist, then the word does not exist and it is inserted as a new node on the left

Binary Search Trees

  - If, on the other hand, the word was greater than the key in the current node, move to the right subtree and compare again

  - If the right subtree does not exist, then the word does not exist and it is inserted as a new node on the right

- This insertion can most easily be effected in a recursive manner

Binary Search Trees

– The point here is that the structure of the tree depends on the order in which the data is inserted in the list

– If the words are entered in sorted order, then the tree will degenerate to a 1-D list

BST Operations

- $Insert$: E x BST $\rightarrow$ BST :

  The function value $Insert(e,T)$ returns the BST T with the element e inserted as a leaf node; if the element already exists, no action is taken

NO WINDOW!!!

BST Operations

- $Delete$: E x BST $\rightarrow$ BST :

  The function value $Delete(e, T)$ returns the BST T with the element e deleted; if the element is not in the BST, no action is taken.

NO WINDOW!!!

Implementation of *Insert*(*e*, *T*)

- If T is empty (i.e. T is NULL)
  - create a new node for e
  - make T point to it
- If T is not empty
  - if e < element at root of T
    - Insert e in left child of T: $Insert(e, T(1))$
  - if e > element at root of T
    - Insert e in right child of T: $Insert(e, T(2))$

Implementation of *Delete*(e, T)

First, we must locate the element e to be deleted in the tree

- if e is at a leaf node
  - we can delete that node and be done
- if e is at an interior node at w
  - we **can't** simply delete the node at w as that would disconnect its children
- if the node at w has only one child
  - we can replace that node with its child

Implementation of *Delete*(*e*, *T*)

– if the node at w has two children

- we replace the node at w with the lowest-valued element among the descendents of its right child

- this is the left-most node of the right tree

- It is useful to have a function DeleteMin() which
  removes the smallest element from a non-empty tree
  and
  returns the value of the element removed

Implementation of *Delete*(*e*, *T*)

- If T is not empty
  - if e < element at root of T

    Delete e from left child of T: *Delete(e, T(1))*
  - if *e* > element at root of T

    Delete *e* from right child of T: *Delete(e, T(2))*
  - if e = element at root of T and both children are empty

    Remove T

Implementation of $Delete(e, T)$

– if e = element at root of T and left child is empty

Replace T with $T(2)$

– if e = element at root of T and right child is empty

Replace T with $T(1)$

– if e = element at root of T and neither child is empty

Replace T with left-most node of $T(2)$ ← *"left-most node in right sub-tree!"*

Implementation of *Delete*(*e*, *T*)

What if the left-most node in the right sub-tree has two (interior node) children?

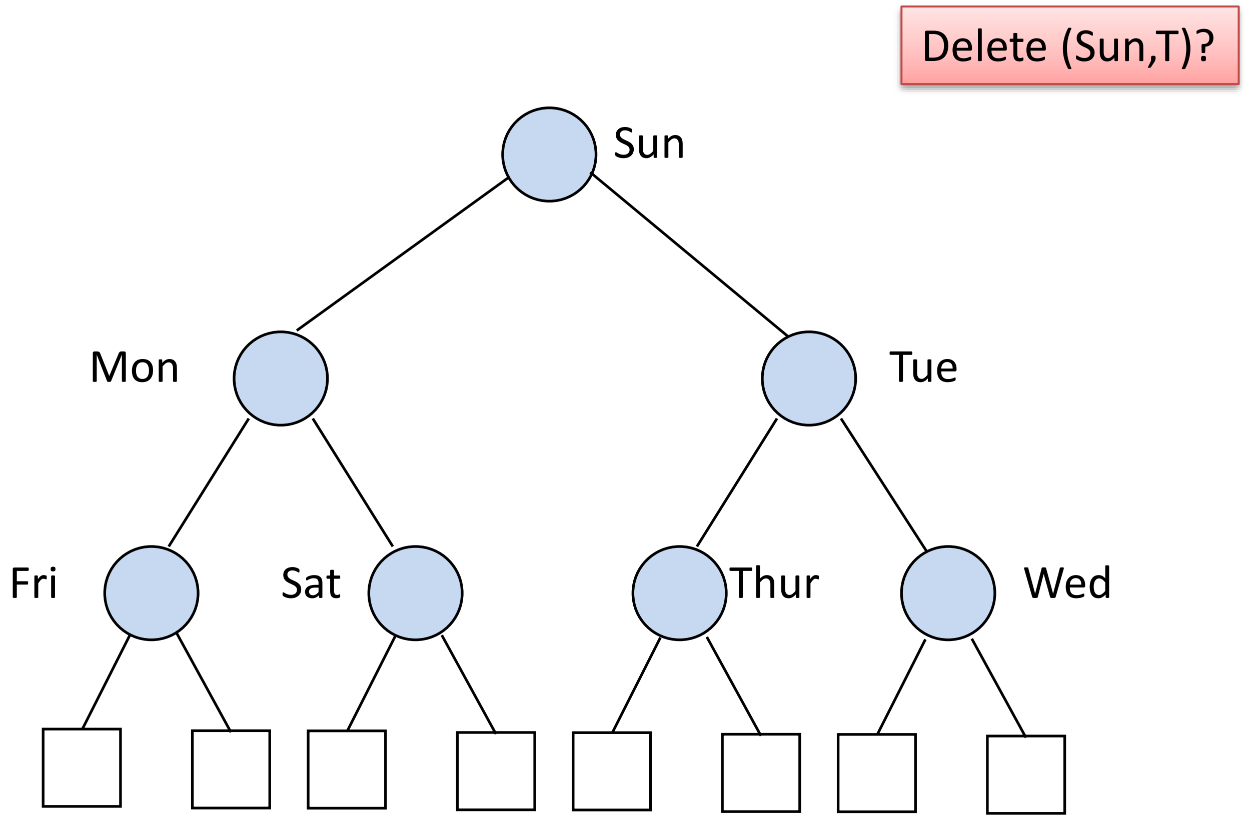
Implementation of *Delete*(*e*, *T*)

It can’t!

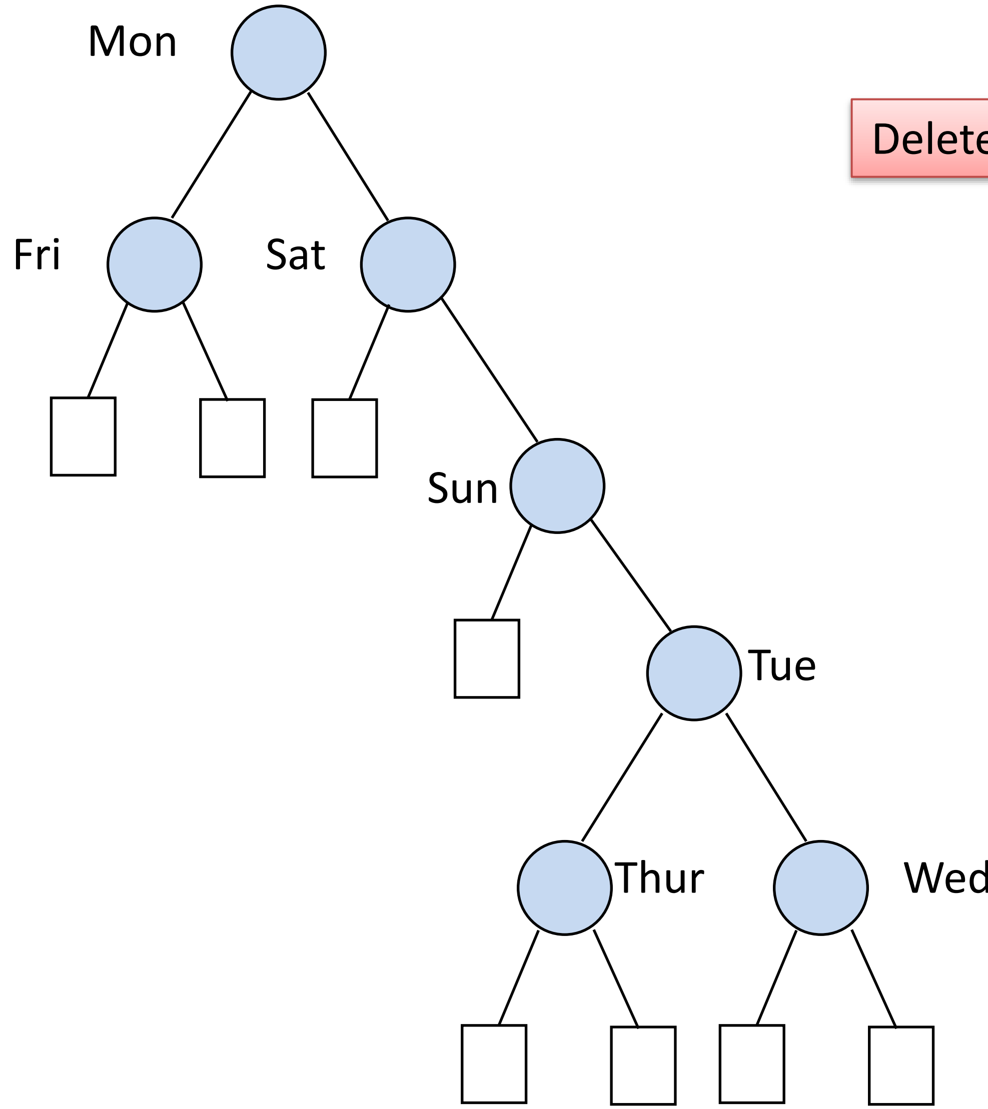
If it did, it wouldn’t be the left-most node …

because there would be a node on it’s left!

Implementation of *Delete*(*e*, *T*)

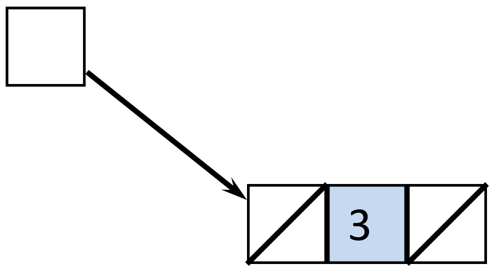


Implementation of *Delete*(*e*, *T*)



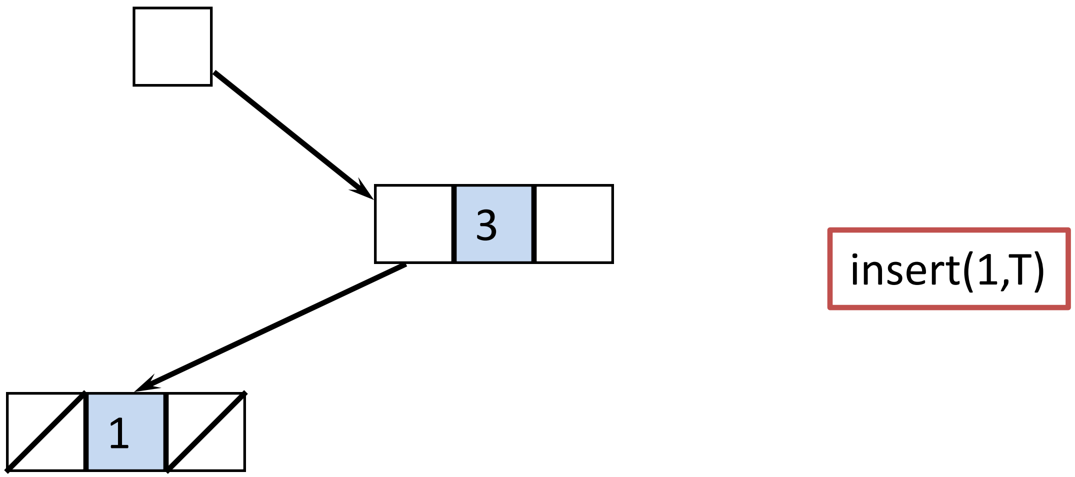
Delete(Mon,T)?

BST Operation: insert

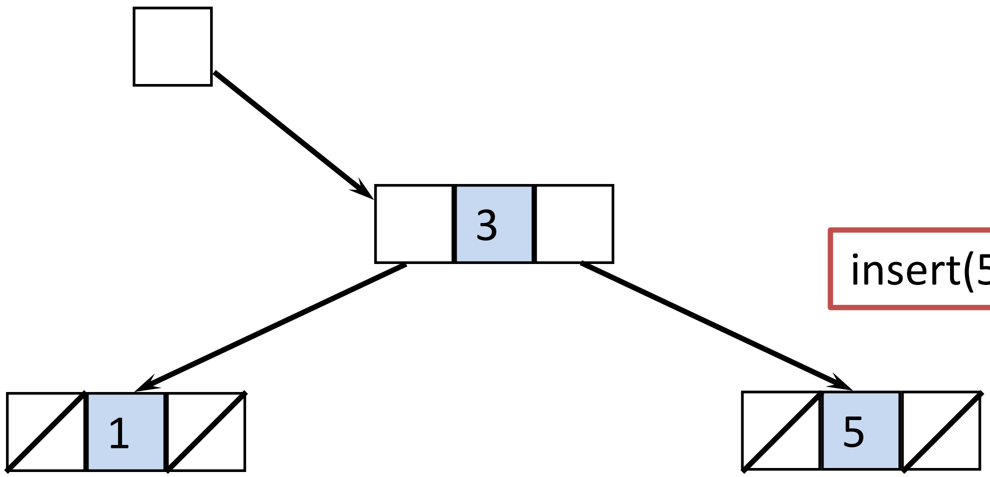


insert(3,T)

BST Operation: insert

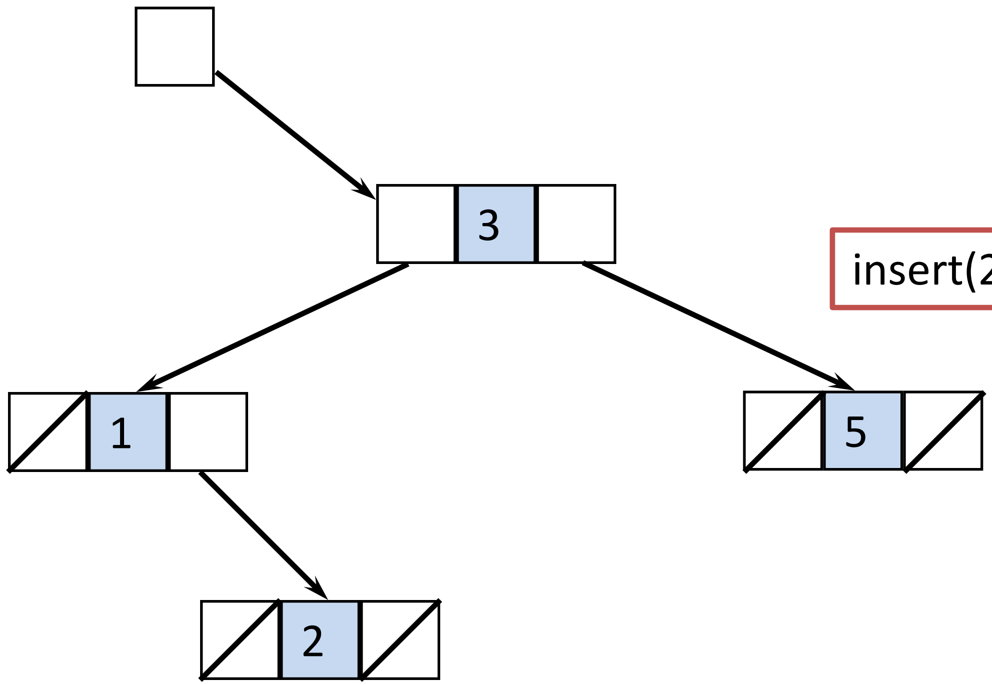


BST Operation: insert



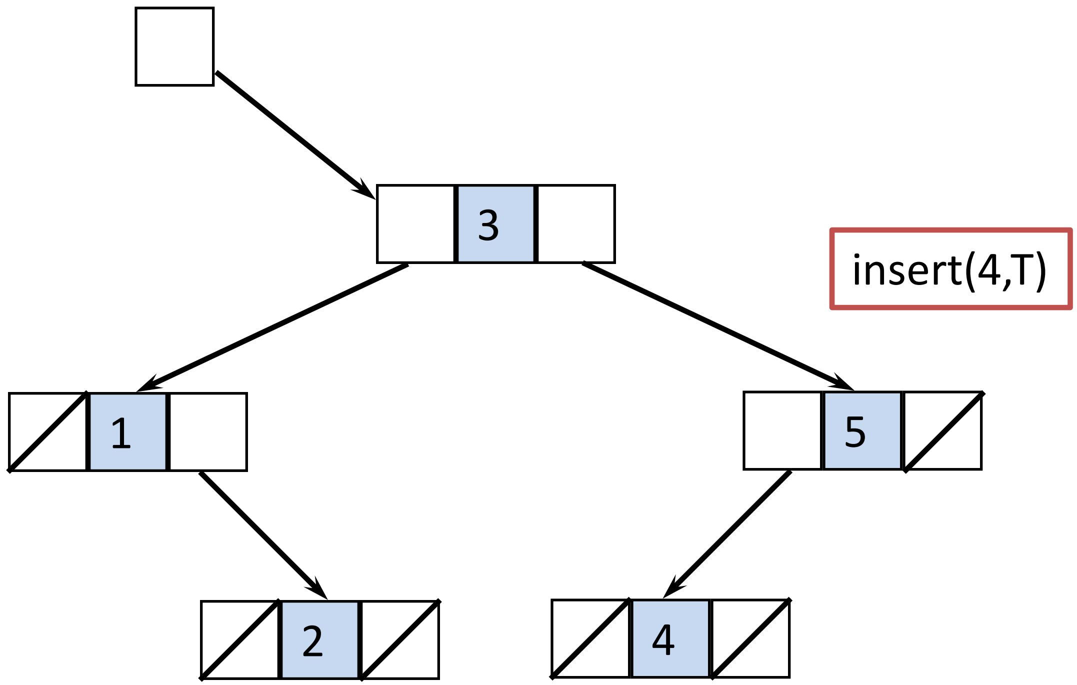
insert(5,T)

BST Operation: insert

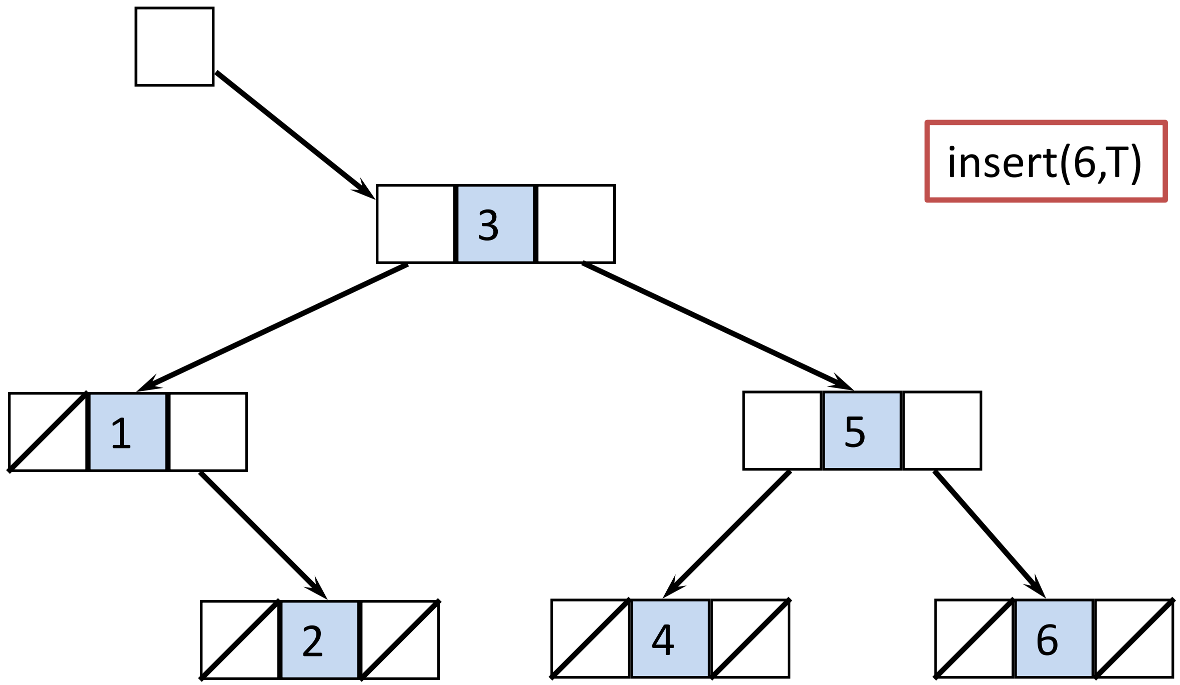


insert(2,T)

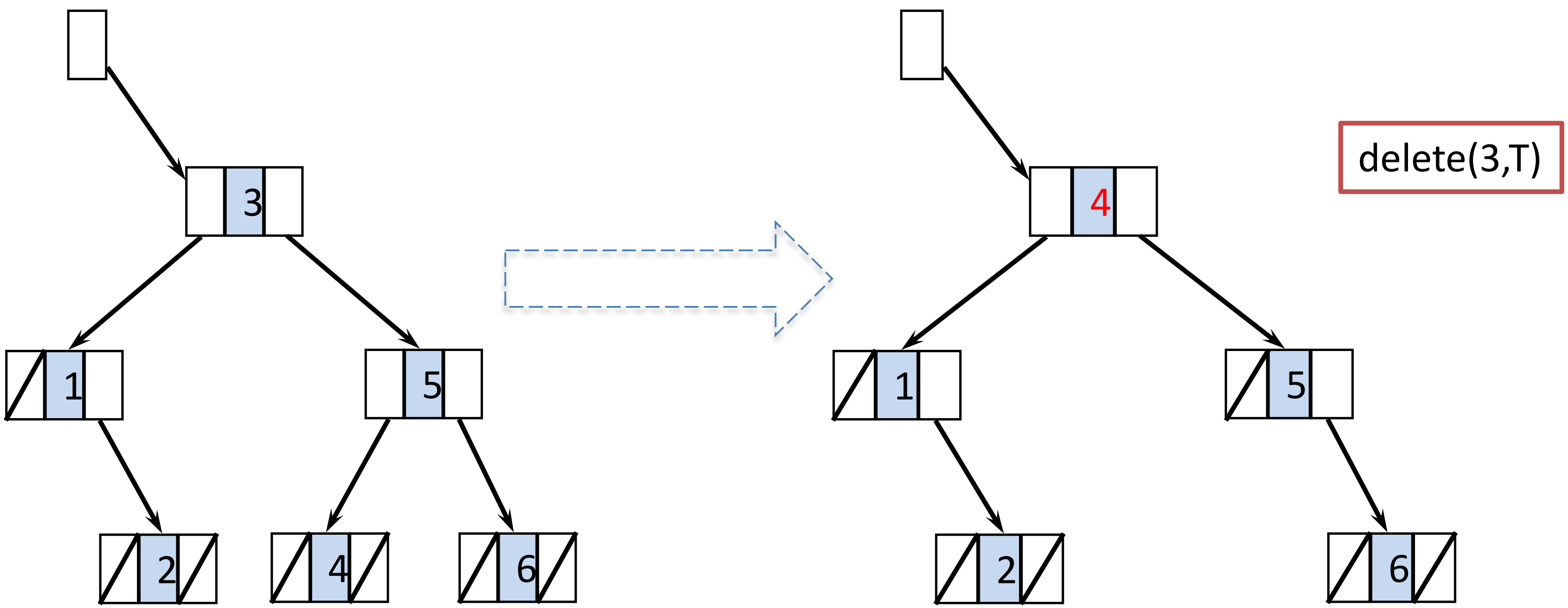
BST Operation: insert



BST Operation: insert



BST Operation: delete



BST: Example Implementation

BST Implementation: BST class

```cpp
#pragma once
#ifndef BS_TREE_H
#define BS_TREE_H
/* This  file constructs the core of a Binary Search Tree and declares the operations.
It is assumed this is a BST of integer values.
Think of each node of a binary search tree (BST) as being a BST. See left and right pointers.
Of course there are alternative implementations. Think about them.
*/
class bs_tree
{
private:
    int item;
    bs_tree *left; //each node is essentially a BST
    bs_tree *right;
public:
    bs_tree();
    bs_tree(int);
    ~bs_tree(){
    //implement appropriate logic here
    };

    //functions-search, insert, delete, and traversal
    bs_tree * search_item(bs_tree *, int);
    bs_tree *insert_item(bs_tree *, int);
    bs_tree *delete_item(bs_tree *, int);
    void in_order(bs_tree *);
    void pre_order(bs_tree *);
    void post_order(bs_tree *);

    //other utility functions
    bs_tree *find_min(bs_tree *);
    bs_tree *find_max(bs_tree *);
};

#endif
```

BST: member functions- constructors

```
#include<iostream>
#include<cstdlib>
using namespace std;

#include"bst.h"
/*
This  file implements the operations and tests some of them.
*/

/*
Define the constructors
*/
bs_tree::bs_tree(){//default constructor
    item=0;
    left=NULL;
    right=NULL;
}

bs_tree::bs_tree(int value){ //parametrized constructor
    item=value;
    left=NULL;
    right=NULL;
}

```

BST: member functions- search_item(e,T)

```
/*
Function returns a pointer to the node a.k.a. tree that has the result
*/
 bs_tree *bs_tree::search_item(bs_tree * root, int e)
 {
     if(root==NULL||root->item==e)
     {
         return root;
     }


     if(e<root->item)
     {
         return search_item(root->left,e);
     }else
     {
         return search_item(root->right,e);
     }
 }
```

BST: member functions- insert (e,T)

```
 /*
Insertion: Perform binary search to determine point of insertion.
Replace the termination NIL pointer with the new item.
Remember that each node has a left and right subtree.
*/

bs_tree *bs_tree::insert_item(bs_tree *root, int value)
{
    if(root==NULL)//if tree is empty
    {
        return new bs_tree(value);
    }
    //otherwise, insert left or right as appropriate
    if(value < root->item)
    {
        root->left=insert_item(root->left,value); //insert left

    }else
    {
        root->right=insert_item(root->right,value); //insert right
    }
    return root;//returns the modified tree after insertion
}
```

BST: member functions- delete(e,T)

```
/*
Deleting from tree is not as straight forward. Multiple cases:
Case-1: node is leaf, just NIL the node's parent pointer.
Case-2: node has one child, just cut the node out,( make the parent of the cdhild to be what was parent of node being deleted).
Case-3: node has both children, relabel the node as its successor and delete the successor.

The function deletes the node and re-arranges the tree.*/

bs_tree *bs_tree::delete_item(bs_tree *root, int item)
{
    if(root==NULL)//for empty tree
        return root;

    //search for node
    if(item<root->item) //search for item left
    {
        root->left=delete_item(root->left,item);
    }
    else if(item>root->item) //search for item right
    {
        root->right=delete_item(root->right,item);
    }
```

BST: member functions- delete(e,T)

```
        else //data is in root
        {
            if(root->left==NULL)//tree only has only one child or no child
            {
                bs_tree *temp=root->right;
                delete root;
                return temp;
            }else if(root->right==NULL)
            {
                bs_tree*temp=root->left;
                delete root;
                return temp;
            }

            //node has both children- get successor, then delete the node
            bs_tree *successor=find_min(root->right);
            //copy inorder successor's content to the current node
            root->item=successor->item;
            //delete the in order successor
            root->right=delete_item(root->right,successor->item);
        }
        return root;
}
```

BST: member functions- find_min(T)

```
/*
Return a pointer to node with minimum value.
The minimum is the left most node.
*/
bs_tree *bs_tree::find_min(bs_tree *root)
{
    bs_tree *min;//pointer to minimum
    if(root==NULL)
    {
        return NULL;
    }

    min=root;//set minimum to current item
    while(min->left!=NULL)
    {
        min=min->left; //progressively go left most
    }
    return min;
}
```

BST: member functions- find_max(T)

```
/*
Return a pointer to node with maximum value.
The maximum is the right most node.
*/
bs_tree *bs_tree::find_max(bs_tree *root)
{
    bs_tree *max;//pointer to maximum
    if(root==NULL)
    {
        return NULL;
    }

    max=root;//set maximum to current item
    while(max->right!=NULL)
    {
        max=max->right; //progressively go right most
    }
    return max;
}
```

BST: member functions- in_order(T), pre_order(T)

```
/*
In-order Traversal.
Procedure: left, root, right
*/
void bs_tree::in_order(bs_tree *root)
{
    if(root!=NULL)
    {
        in_order(root->left);
        cout<<root->item<<" ";
        in_order(root->right);
    }
}
/*
Pre-order Traversal.
Procedure: root, left, right
*/
void bs_tree::pre_order(bs_tree *root)
{
    if(root!=NULL)
    {
        cout<<root->item<<" ";
        pre_order(root->left);
        pre_order(root->right);
    }
}
```

BST: member functions- post_order(T)

```
 /*
Post-order Traversal.
Procedure: left, right, then root
*/
void bs_tree::post_order(bs_tree *root)
{
    if(root!=NULL)
    {
        post_order(root->left);
        post_order(root->right);
        cout<<root->item<<" ";

    }
}
```

Driver program: main()

```
/*
main function to set up a BST and call its member functions.
*/

int main()
{
    bs_tree bst, *root=NULL;
    /* BST example
                    12
                   /    \
                2        25
               /    \    /
              1    3    14
    */
    root=bst.insert_item(root,12);
    root=bst.insert_item(root, 2);
    root=bst.insert_item(root, 3);
    root=bst.insert_item(root, 25);
    root=bst.insert_item(root, 14);
    root=bst.insert_item(root, 1);
    cout<<"In Order Traversal"<<endl;
    bst.in_order(root);
    cout<<endl;
    cout<<"Pre Order Traversal"<<endl;
    bst.pre_order(root);
    cout<<endl;
    cout<<"Post Order Traversal"<<endl;
    bst.post_order(root);
    cout<<endl;
```

Driver program: main()

```
        //delete node 1
            /* BST - after deleting 1.
                            12
                          /    \
                        2        25
                          \    /
                            3    14
        */
        cout<<"Node "<<1<<" deleted."<<endl;
        root=bst.delete_item(root,1);
        bst.post_order(root);
        //delete node 12
            /* BST - after deleting 12.
                            14
                          /    \
                        2        25
                        \
                        3
        */
         cout<<"Node "<<12<<" deleted."<<endl;
         root=bst.delete_item(root,12);
         bst.pre_order(root);


         //search for 2
         bs_tree *node=bst.search_item(root,2);
         if(node!=NULL)
             cout<<"\n"<<2<<" Found!!"<<endl;
         else
             cout<<"\n"<<2<<" Not found!!"<<endl;

        system("pause");
        return 0;
}
```

Applications of BST

- Implementation of searching algorithms
- Implementation of sorting algorithms:
  - Elements are added and traversed using in-order traversal.
- Indexing and multi-level indexing
- etc.