04-630
Data Structures and Algorithms for Engineers

David Vernon
Carnegie Mellon University Africa

vernon@cmu.edu
www.vernon.eu

Lecture 13

Trees I

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

Binary Search Trees

- A Binary Search Tree (BST) is a special type of binary tree
  - it represents information is an ordered format
  - A binary tree is binary search tree if for every node w,

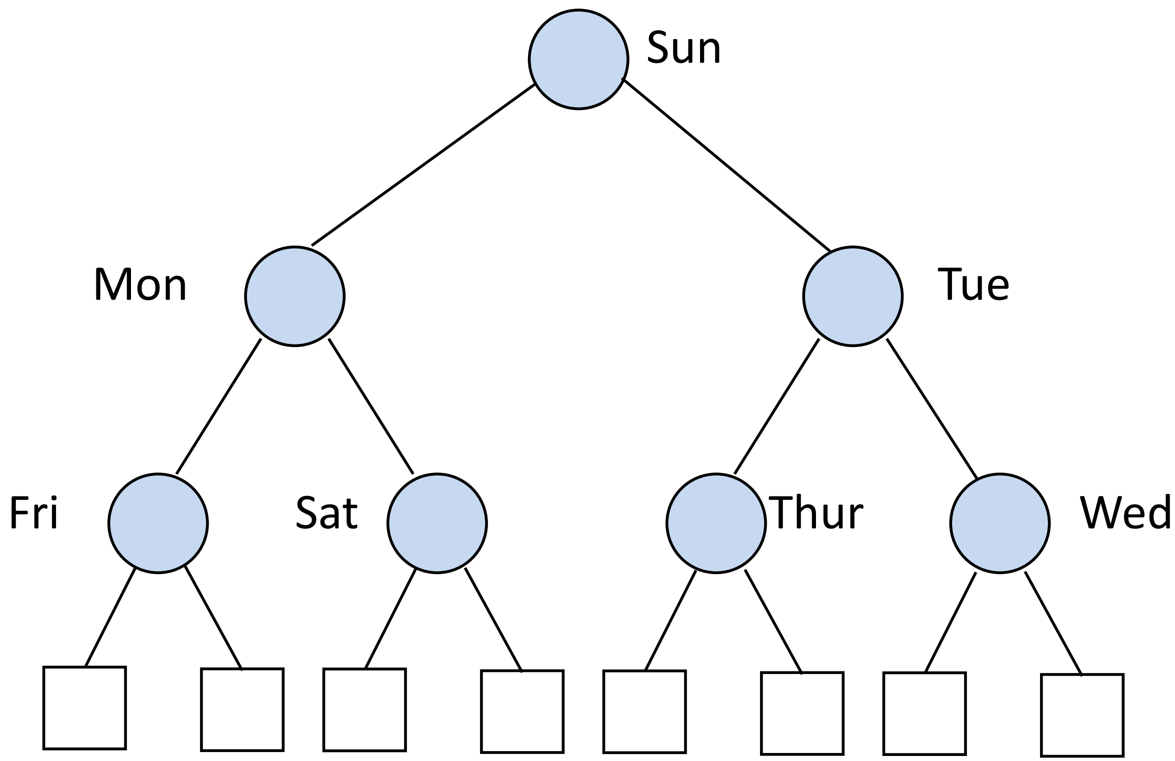
    all keys in the left subtree of i have values less than the key of w and

    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 identifers 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

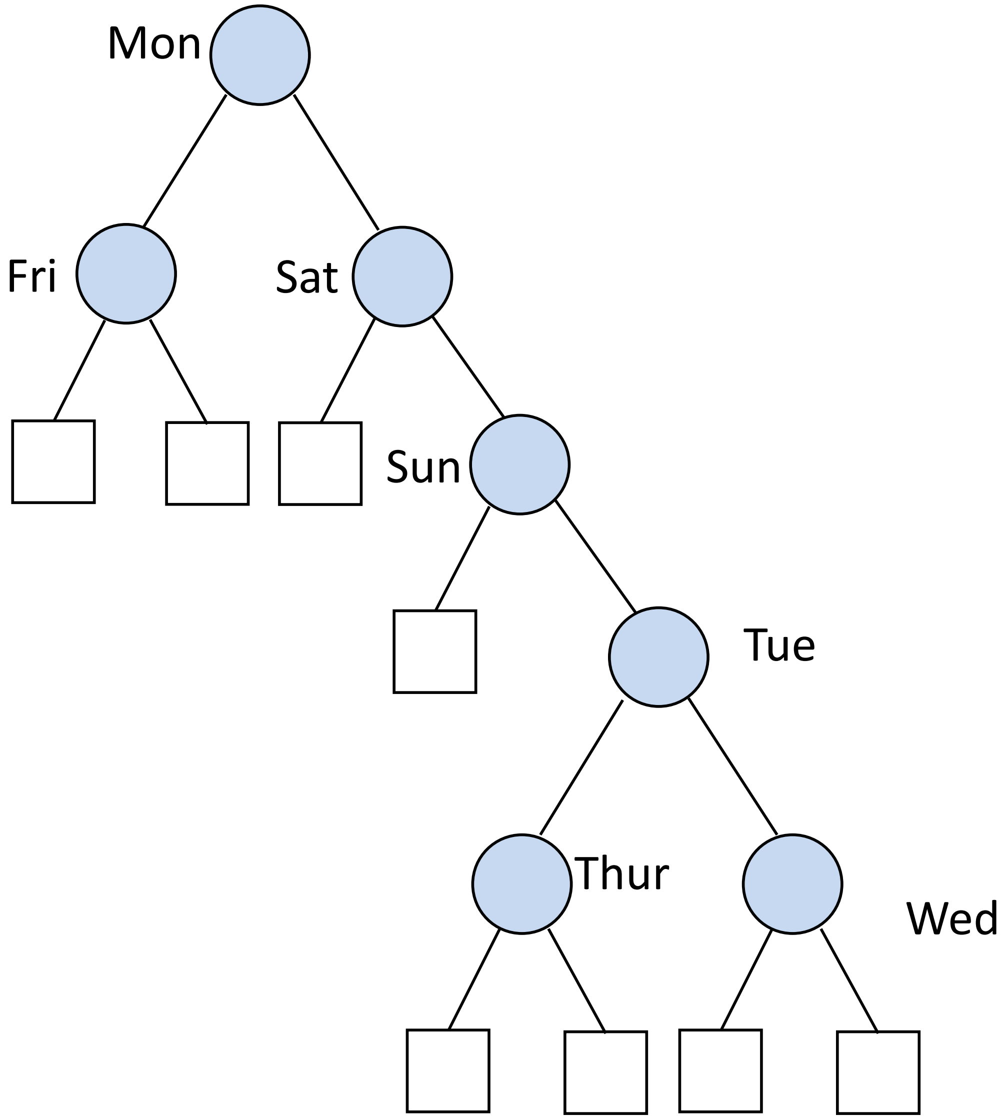
- Furthermore, efficient searching is possible using the binary search technique

  – search time is $O(log_2n)$

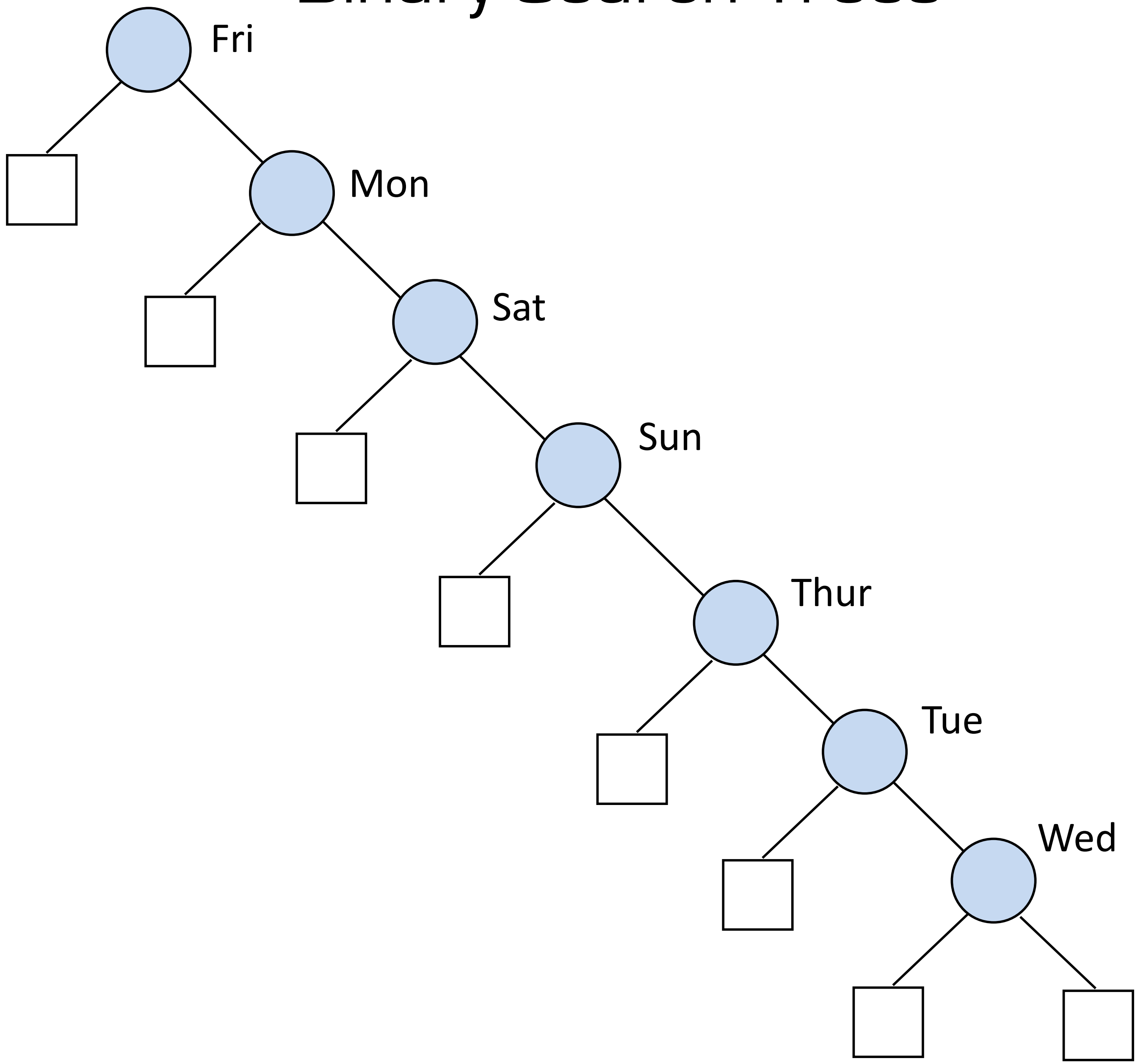
Binary Search Trees

- 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

  To do so, we need to address how a binary search tree is constructed

  – 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 $Insert(e,T)$ is 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 $Delete(e, T)$ is the BST T with the element e deleted; if the element is not in the BST exists, 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 with removes the smallest element from a non-empty tree and returns the value of the element removed

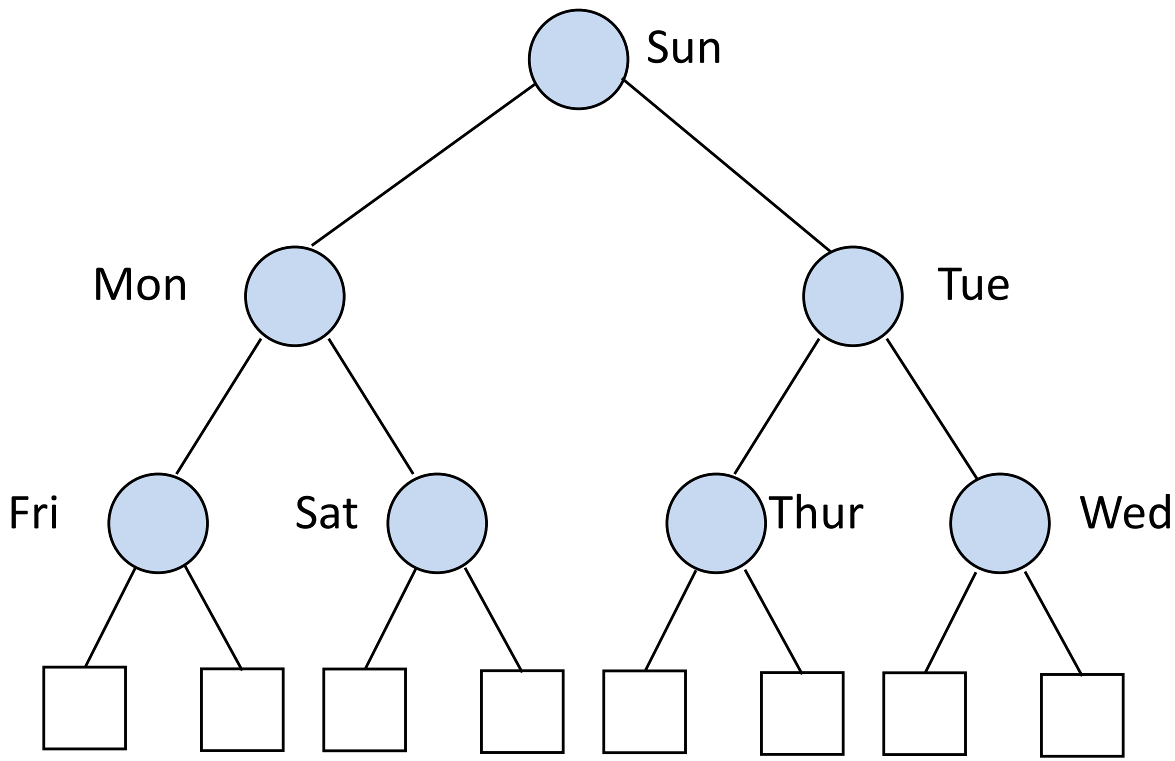
Implementation of $Delete(e, T)$

- 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
  - if e = element at root of T and left child is empty
    - Replace T with $T(2)$

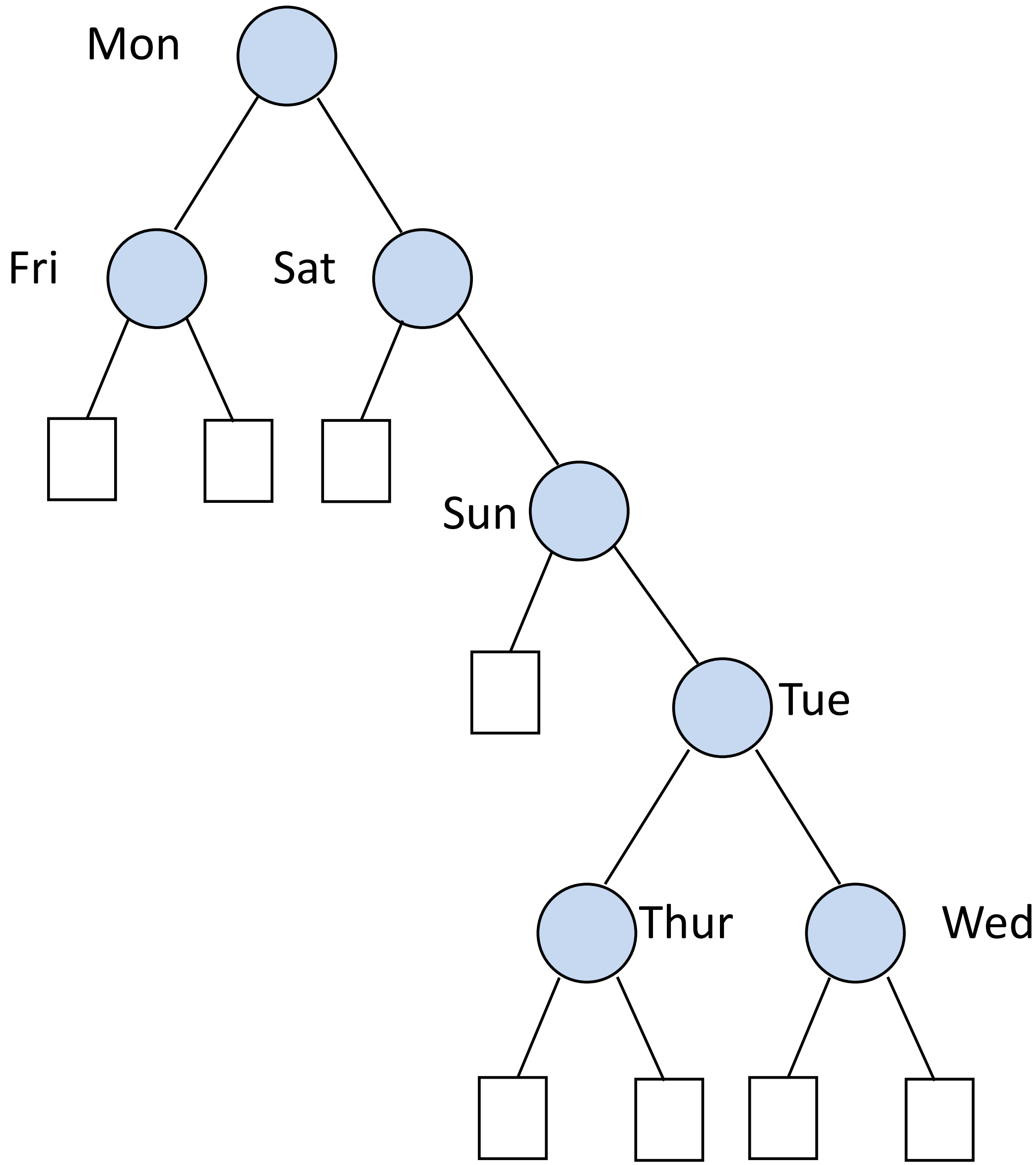
Implementation of $Delete(e, T)$

- 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 $T(2)$

Implementation of $Delete(e, T)$



Implementation of $Delete(e, T)$



Tree Traversals

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

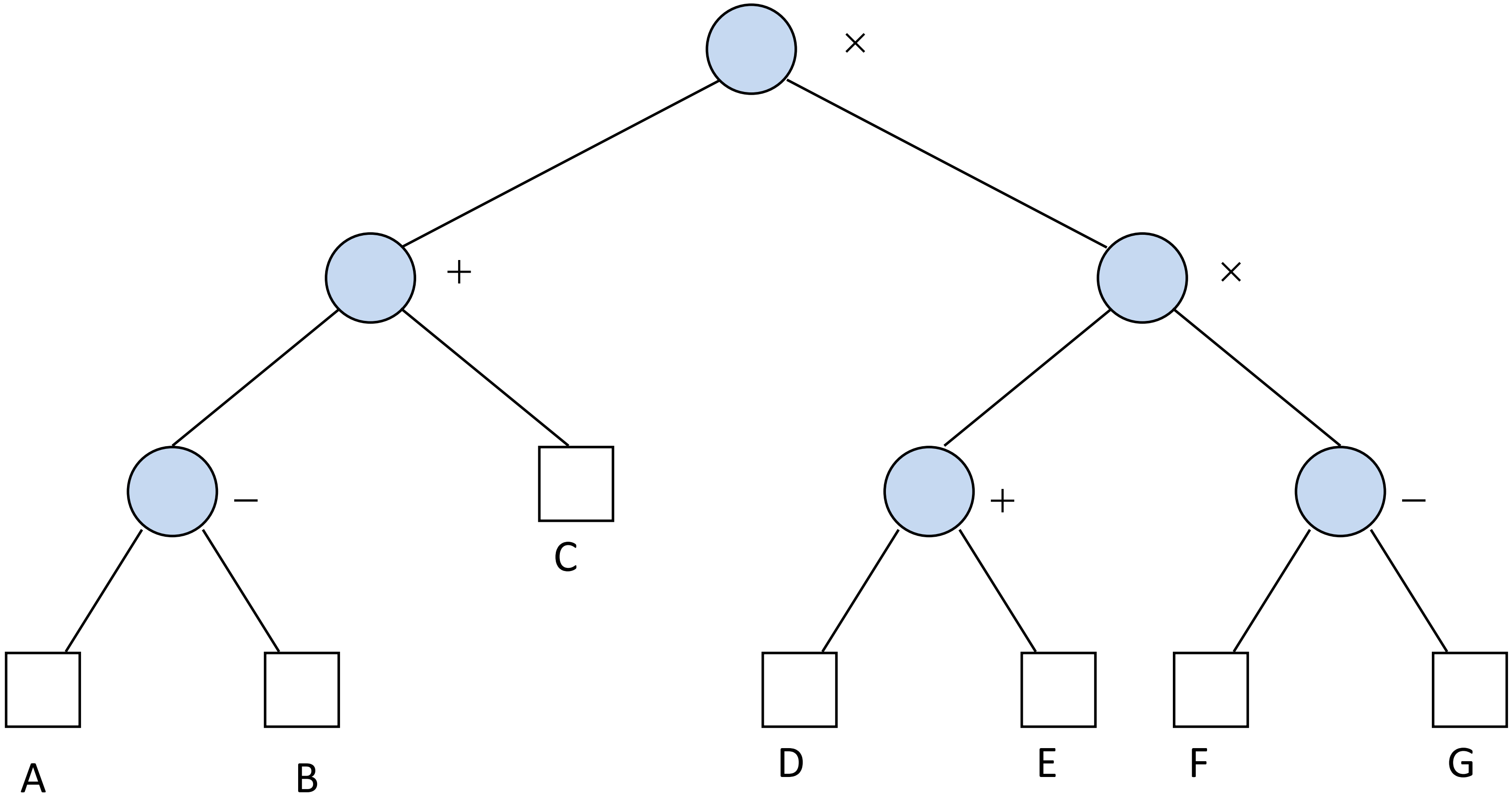
- Traversals can be used

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

Depth-First Traversals

- There are 3 depth-first traversals
  - Inorder
  - Postorder
  - Preorder
- 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

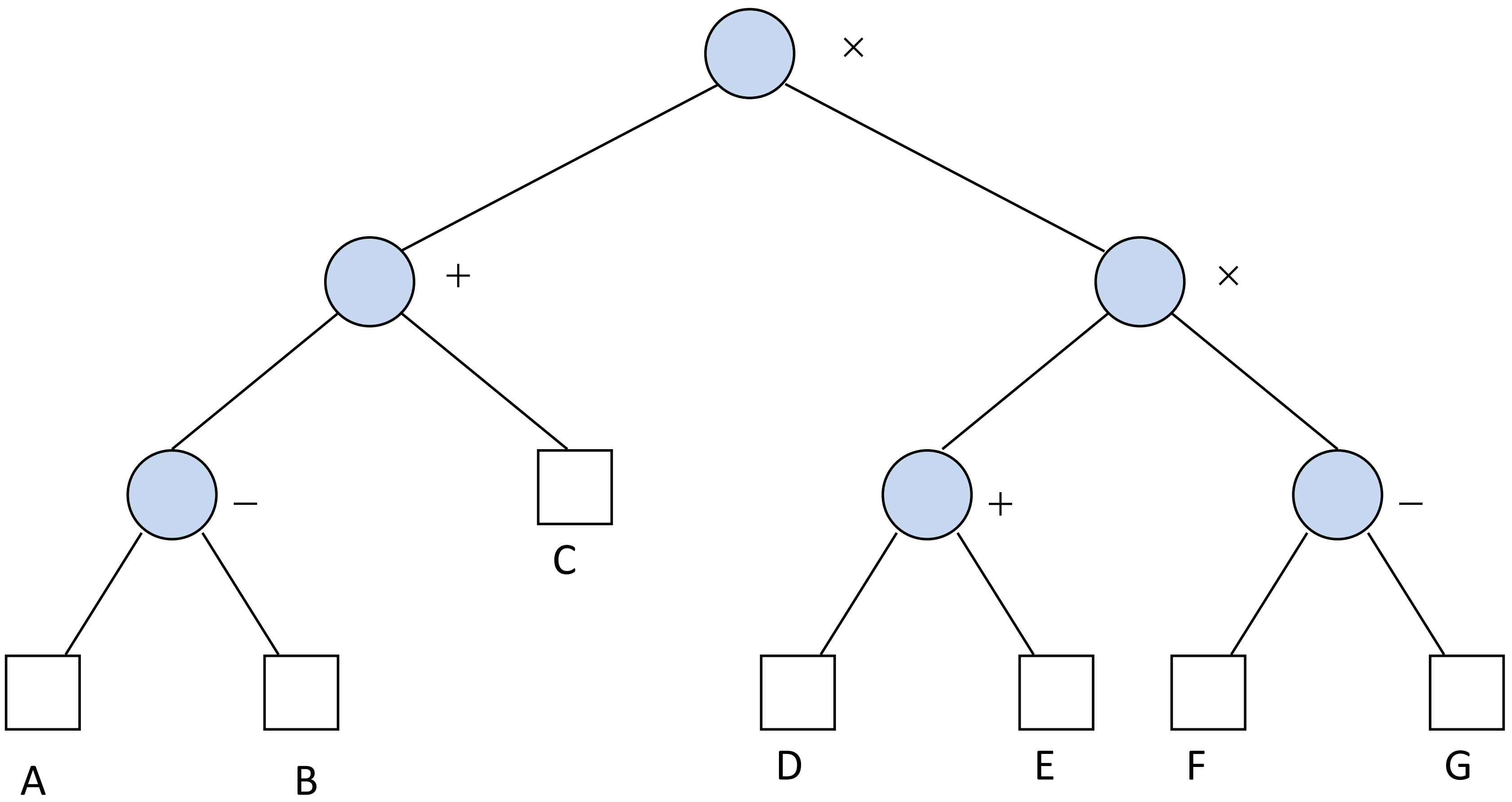
  otherwise

  perform an inorder traversal of $Left(T)$

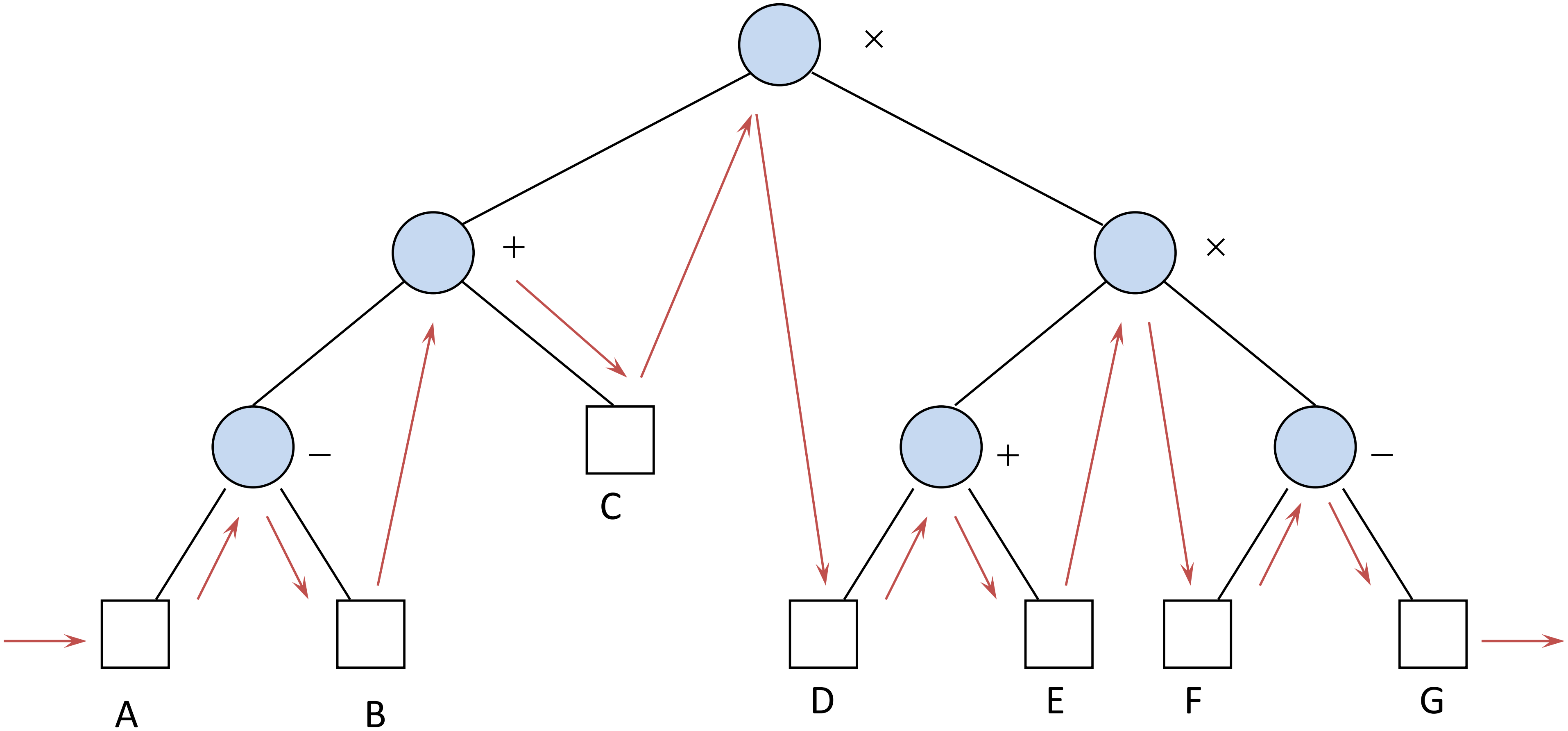
  visit the root of T

  perform an inorder traversal of $Right(T)$

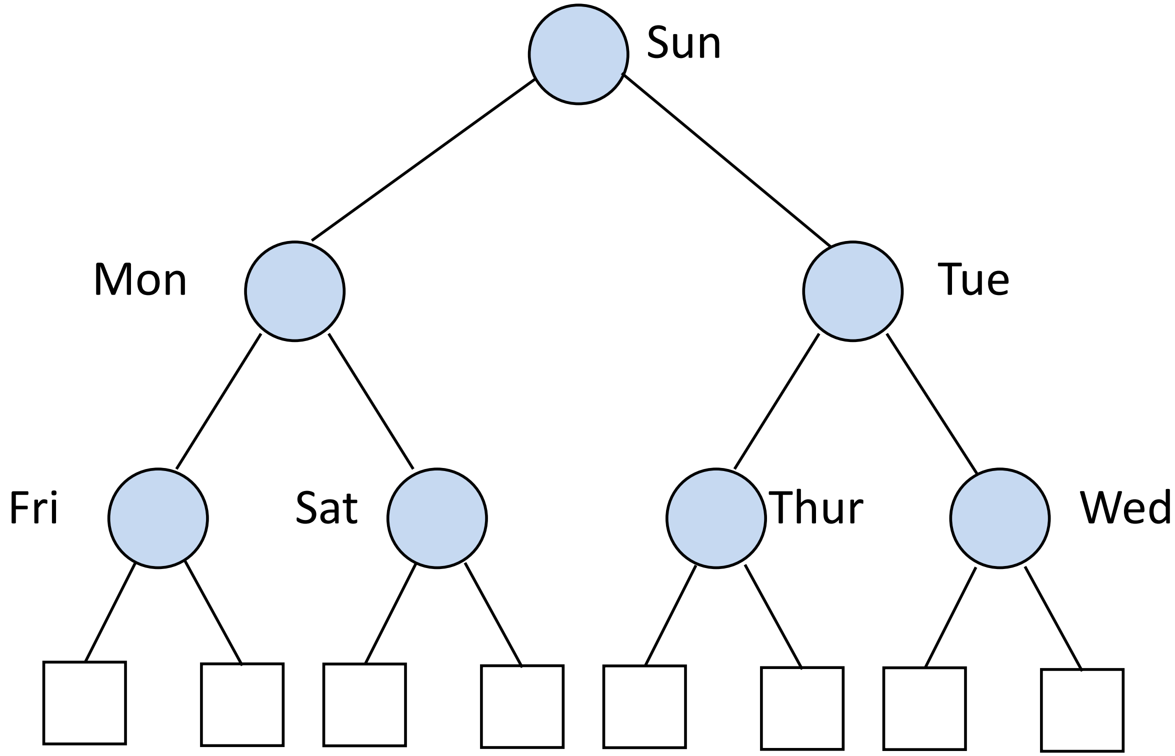
Example: Inorder Traversal



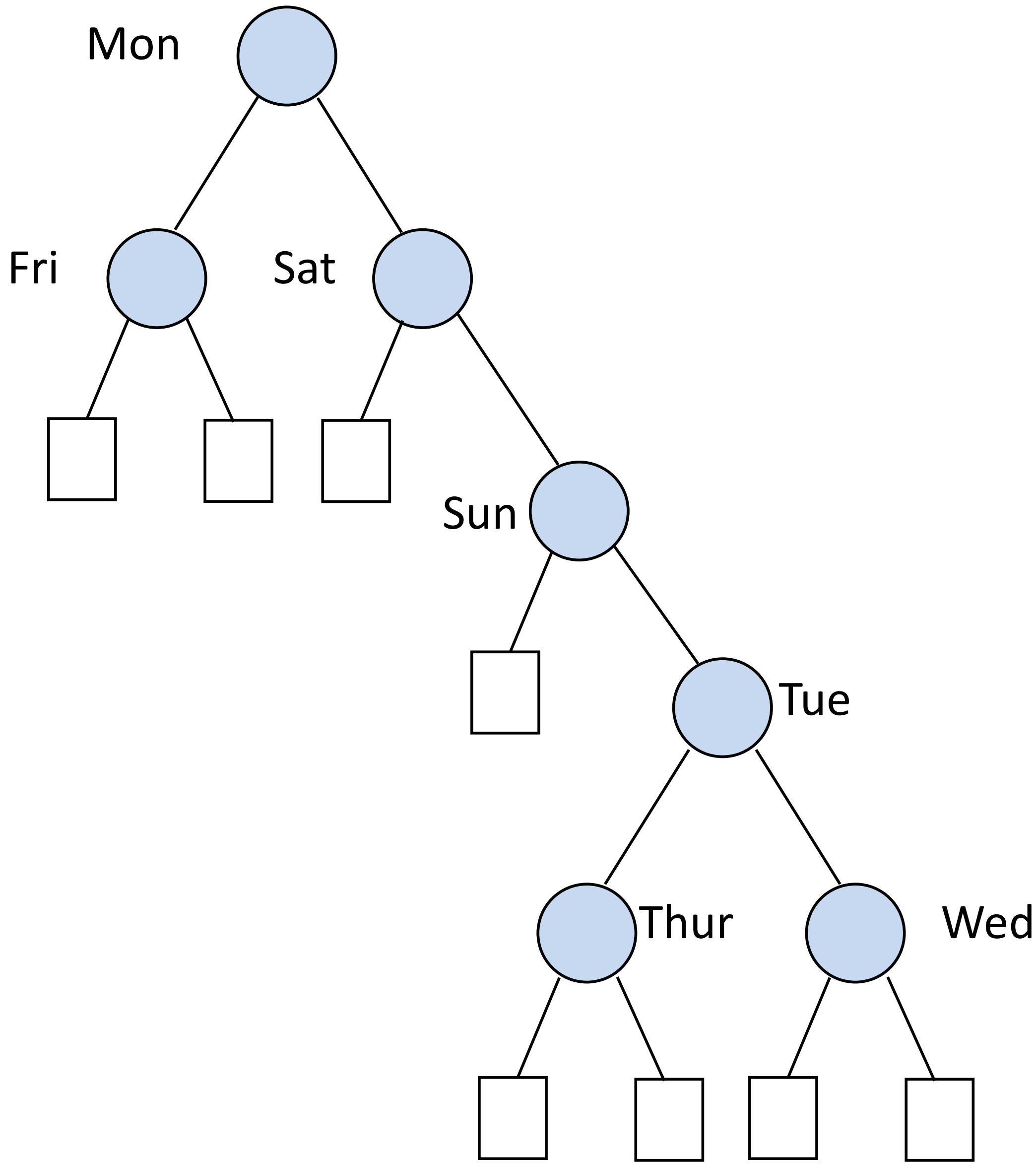
Example: Inorder Traversal



Example: Inorder Traversal



Example: Inorder Traversal



Depth-First Traversals

- Recursive definition of postorder traversal

  Given a binary tree T

  if T is empty

  visit the external node

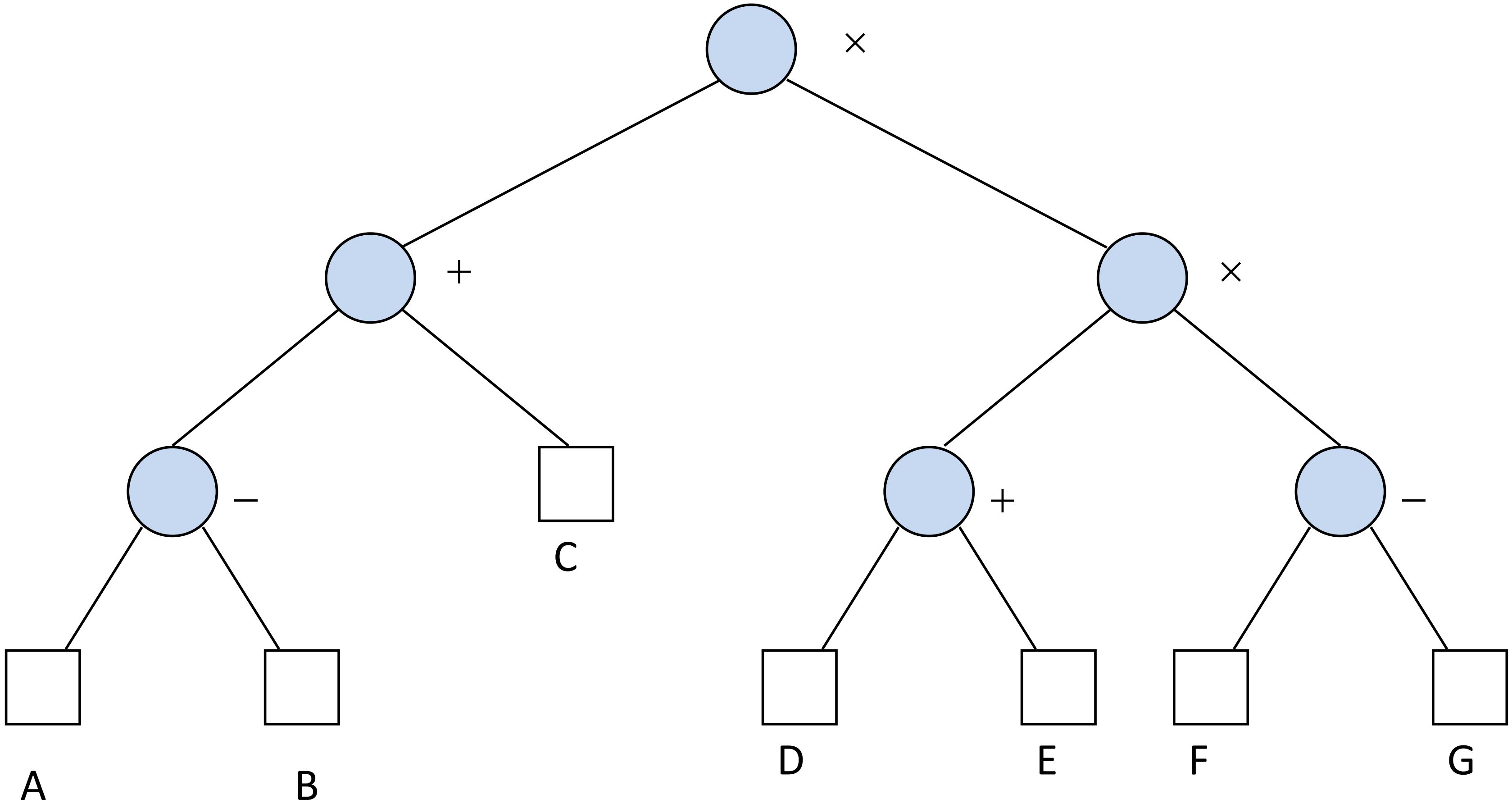
  otherwise

  perform an postorder traversal of $Left(T)$

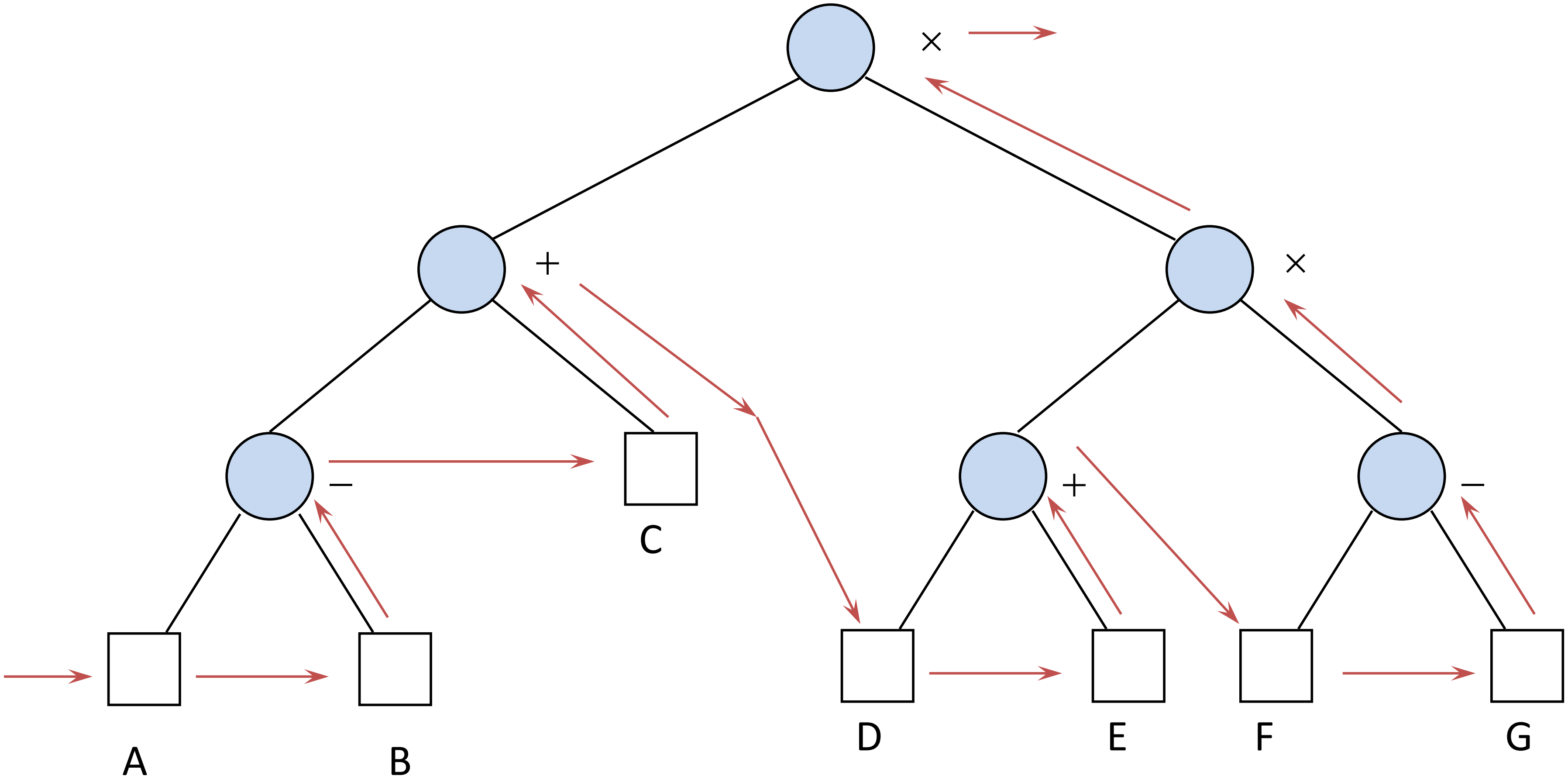
  perform an postorder traversal of $Right(T)$

  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

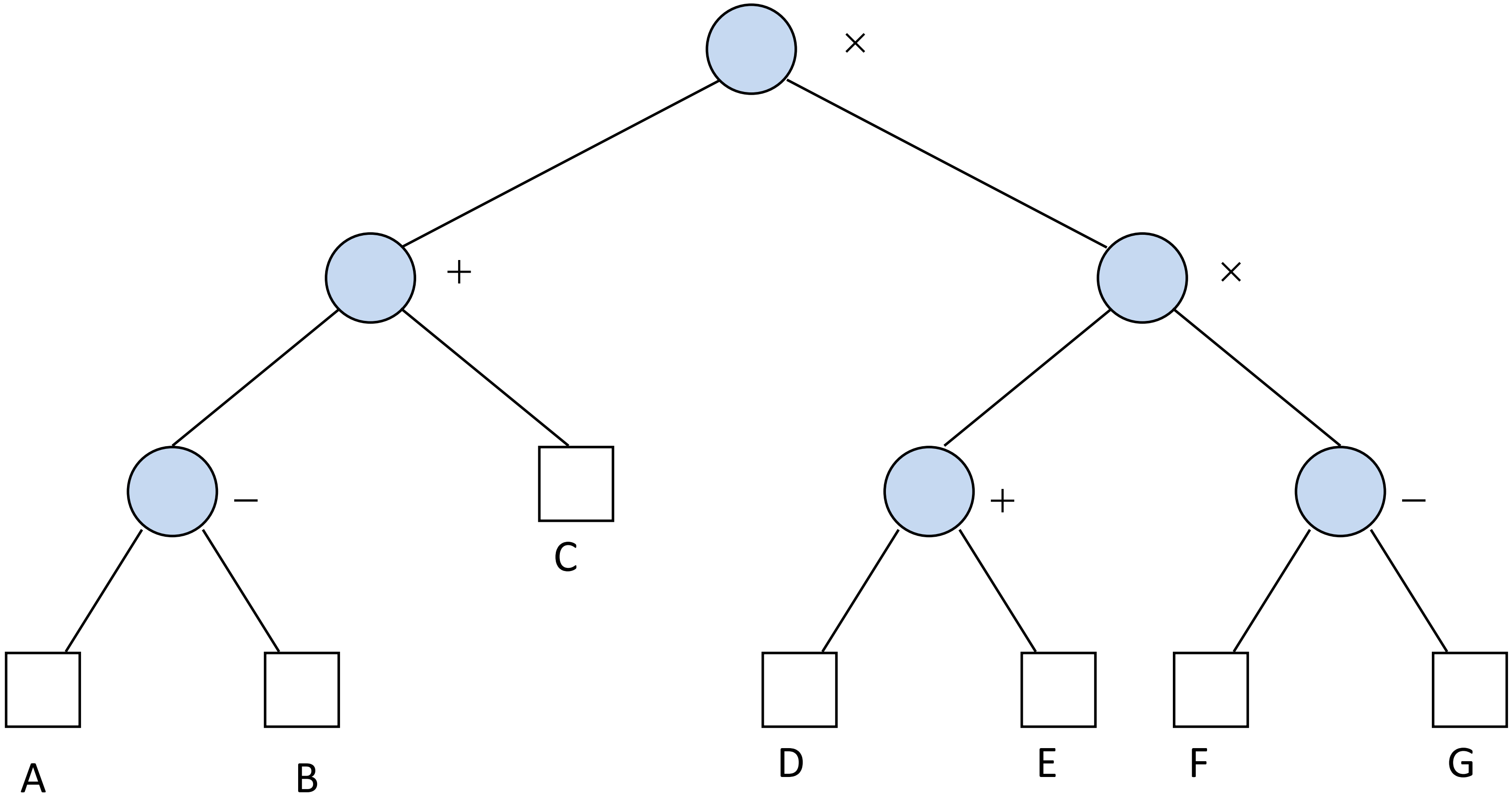
  otherwise

  visit the root of T

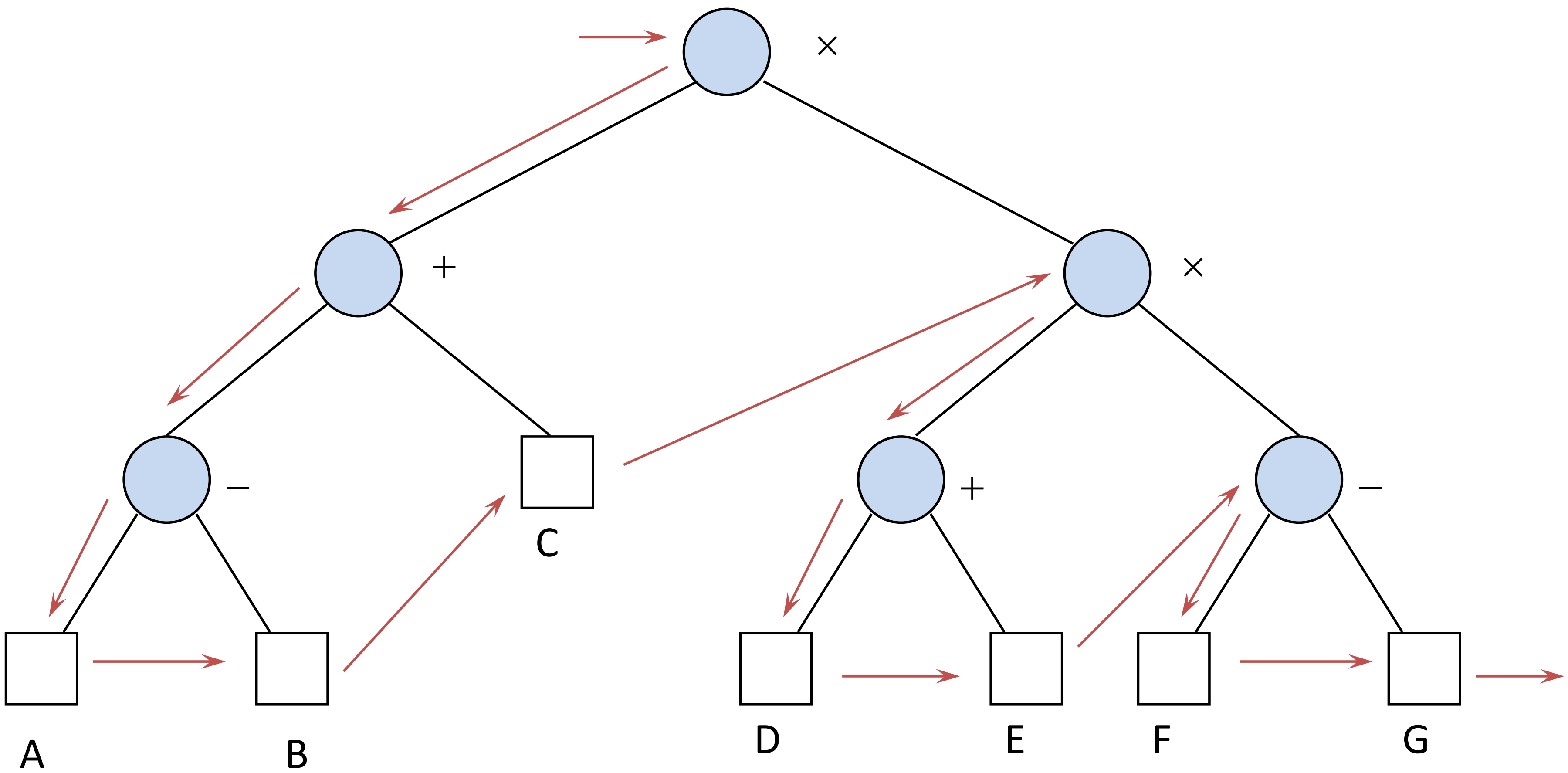
  perform an preorder traversal of $Left(T)$

  perform an preorder traversal of $Right(T)$

Example: Preorder Traversal



Example: Preorder Traversal



BST Implementation

```
typedef  struct {
            int number;
            char *string;
         } ELEMENT_TYPE;

typedef struct node *NODE_TYPE;

typedef struct node {
            ELEMENT_TYPE element;
            NODE_TYPE left, right;
         } NODE;

typedef NODE_TYPE BINARY_TREE_TYPE;

typedef BINARY_TREE_TYPE WINDOW_TYPE;
```

```c
int main() {

   ELEMENT_TYPE e;
   BINARY_TREE_TYPE tree;

   initialize(&tree);

   print(tree);

   assign_element_values(&e, 3, "...");
   insert(e, &tree);
   print(tree);

   assign_element_values(&e, 1, "+++");
   insert(e, &tree);
   print(tree);

   assign_element_values(&e, 5, "---");
   insert(e, &tree);
   print(tree);

   assign_element_values(&e, 2, ";;;");
   insert(e, &tree);
   print(tree);

   assign_element_values(&e, 4, "***");
   insert(e, &tree);
   print(tree);

   assign_element_values(&e, 6, "000");
   insert(e, &tree);
   print(tree);

   assign_element_values(&e, 3, "...");
   delete_element(e, &tree);
   print(tree);
```

```c
/*** initialize a tree ***/

void initialize(BINARY_TREE_TYPE *tree) {

   static bool first_call = true;

   /* we don't know what value *tree has when the program is launched      */
   /* so we have to be careful not to dereference it                       */
   /* if it's the first call to initialize, there is no tree to be deleted */
   /* and we just set *tree to NULL                                        */

   if (first_call) {
      first_call = false;
      *tree = NULL;
   }
   else {
      if (*tree != NULL) postorder_delete_nodes(*tree);
      *tree = NULL;
   }
}
```

```
/*** insert an element in a tree ***/

BINARY_TREE_TYPE *insert(ELEMENT_TYPE e,  BINARY_TREE_TYPE *tree ) {

   WINDOW_TYPE temp;

   if (*tree == NULL) {

      /* we are at an external node: create a new node and insert it */

      if ((temp = (NODE_TYPE) malloc(sizeof(NODE))) == NULL)
         error("function insert: unable to allocate memory");
      else {
         temp->element = e;
         temp->left    = NULL;
         temp->right   = NULL;
         *tree = temp;
      }
   }
   else if (e.number < (*tree)->element.number) { /* assume the number field is the key */
      insert(e, &((*tree)->left));
   }
   else if (e.number > (*tree)->element.number) {
      insert(e, &((*tree)->right));
   }

   /* if e.number == (*tree)->element.number, e already is in the tree so do nothing */

   return(tree);
}
```

```c
/*** returns & deletes the smallest node in a tree (i.e. the left-most node) */

ELEMENT_TYPE delete_min(BINARY_TREE_TYPE *tree) {

   ELEMENT_TYPE e;
   BINARY_TREE_TYPE p;

   if ((*tree)->left == NULL) {

      /* tree points to the smallest element */

      e = (*tree)->element;

      /* replace the node pointed to by tree by its right child */

      p = *tree;
      *tree = (*tree)->right;
      free(p);

      return(e);
   }
   else {

      /* the node pointed to by tree has a left child */

      return(delete_min(&((*tree)->left)));
   }

}
```

```
/*** delete an element in a tree ***/

BINARY_TREE_TYPE *delete_element(ELEMENT_TYPE e, BINARY_TREE_TYPE *tree) {

   BINARY_TREE_TYPE p;

   if (*tree != NULL) {

      if (e.number < (*tree)->element.number)  /* assume element.number is the */
         delete_element(e, &((*tree)->left));  /* key                          */

      else if (e.number > (*tree)->element.number)
         delete_element(e, &((*tree)->right));

      else if (((*tree)->left == NULL) && ((*tree)->right == NULL)) {

         /* leaf node containing e - delete it */

         p = *tree;
         free(p);
         *tree = NULL;
      }
```

```c
    else if ((*tree)->left == NULL) {

       /* internal node containing e and it has only a right child */
       /* delete it and make treepoint to the right child          */

       p = *tree;
       *tree = (*tree)->right;
       free(p);
    }
    else if ((*tree)->right == NULL) {

       /* internal node containing e and it has only a left child */
       /* delete it and make treepoint to the left child          */

       p = *tree;
       *tree = (*tree)->left;
       free(p);
    }
    else {

       /* internal node containing e and it has both left and right child */
       /* replace it with leftmost node of right sub-tree                 */
       (*tree)->element = delete_min(&((*tree)->right));
    }
  }
  return(tree);
}
```

```c
/*** inorder traversal of a tree, printing node elements **/

int inorder(BINARY_TREE_TYPE tree, int n) {

   int i;

   if (tree != NULL) {
      inorder(tree->left, n+1);

      for (i=0; i<n; i++) printf("        ");
      printf("%d %s\n", tree->element.number, tree->element.string);

      inorder(tree->right, n+1);
   }
   return(0);
}
```

```c
/*** inorder traversal of a tree, deleting node elements **/

int postorder_delete_nodes(BINARY_TREE_TYPE tree) {

   if (tree != NULL) {
      postorder_delete_nodes(tree->left);
      postorder_delete_nodes(tree->right);
      free(tree);
   }
   return(0);
}
```

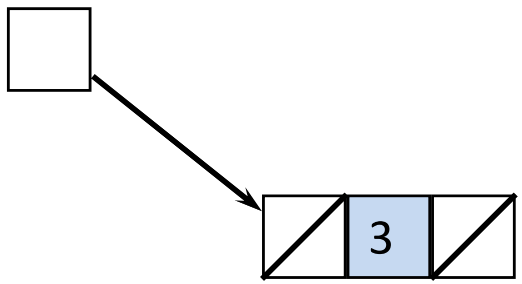
```c
/*** print all elements in a tree by traversing inorder ***/

int print(BINARY_TREE_TYPE tree) {

   printf("Contents of tree by inorder traversal: \n");

   inorder(tree,0);

   printf("--- \n");

   return(0);
}
```

```c
/*** error handler:
     print message passed as argument and take appropriate action ***/

int error(char *s) {

   printf("Error: %s\n",s);

   exit(0);
}
```

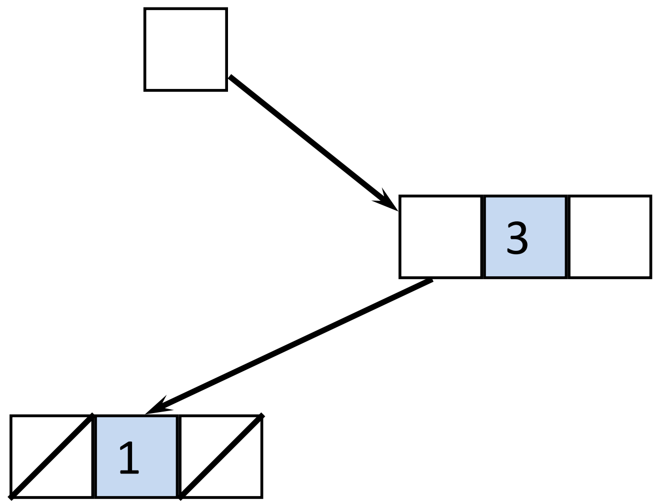
```c
/*** assign values to an element ***/

int assign_element_values(ELEMENT_TYPE *e, int number, char s[]) {

   e->string = (char *) malloc(sizeof(char) * (strlen(s)+1));
   strcpy(e->string, s);
   e->number = number;
   return(0);
}
```

```c
int main() {

   ELEMENT_TYPE e;
   BINARY_TREE_TYPE tree;

   initialize(&tree);

   print(tree);

   assign_element_values(&e, 3, "...");
   insert(e, &tree);
   print(tree);

   assign_element_values(&e, 1, "+++");
   insert(e, &tree);
   print(tree);

   assign_element_values(&e, 5, "---");
   insert(e, &tree);
   print(tree);

   assign_element_values(&e, 2, ";;;");
   insert(e, &tree);
   print(tree);

   assign_element_values(&e, 4, "***");
   insert(e, &tree);
   print(tree);

   assign_element_values(&e, 6, "000");
   insert(e, &tree);
   print(tree);

   assign_element_values(&e, 3, "...");
   delete_element(e, &tree);
   print(tree);
```

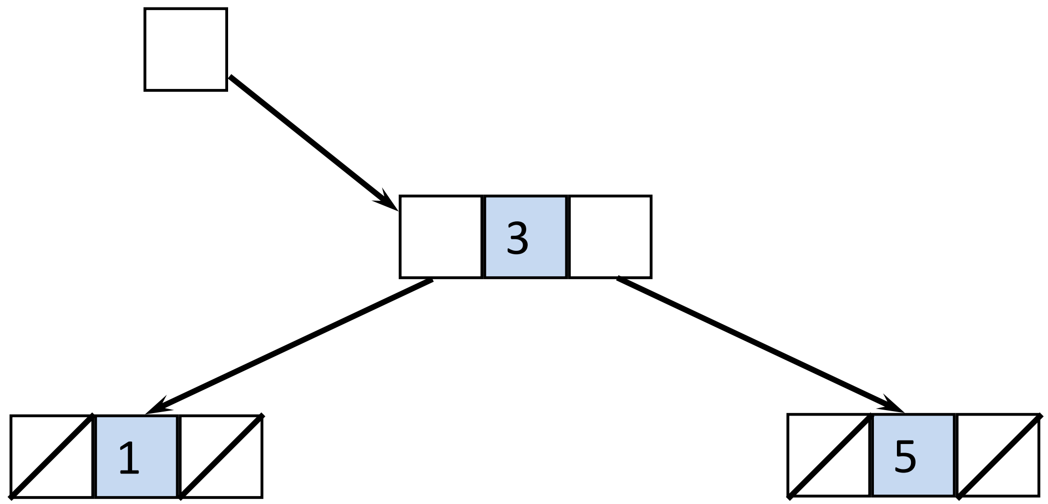
BINARY_TREE Implementation



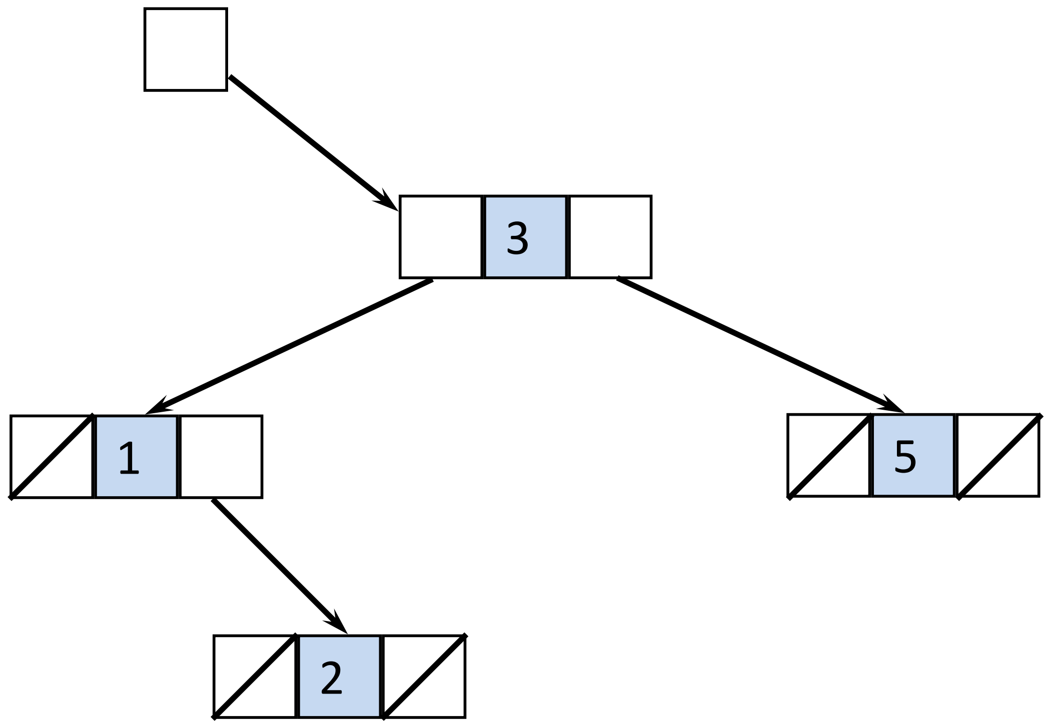
BINARY_TREE Implementation



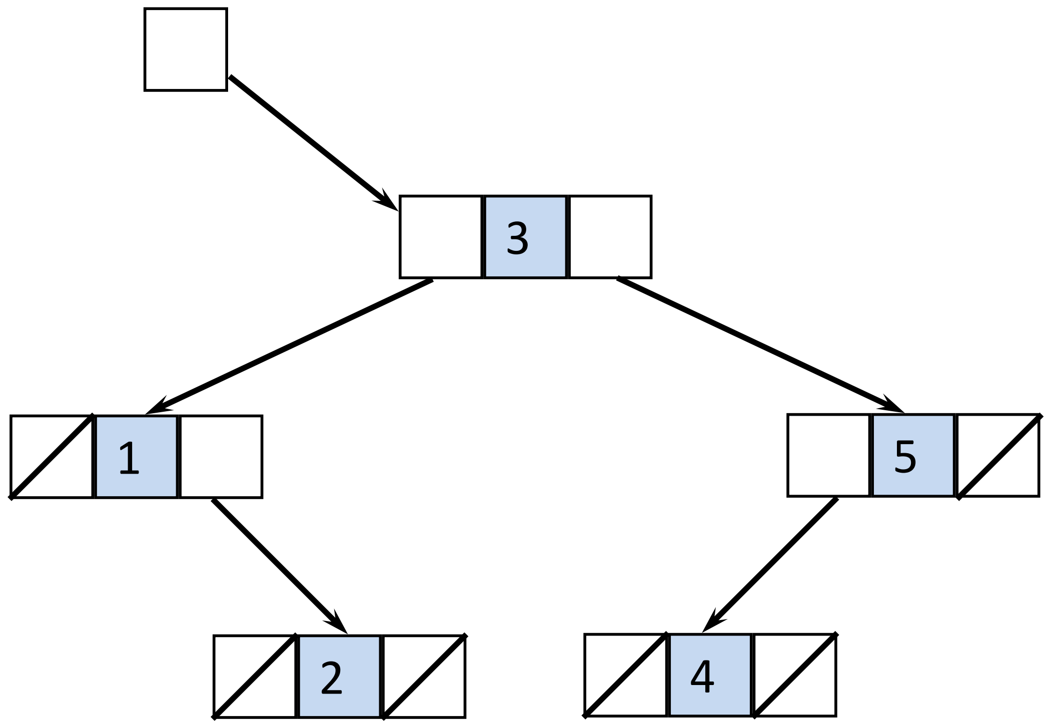
BINARY_TREE Implementation



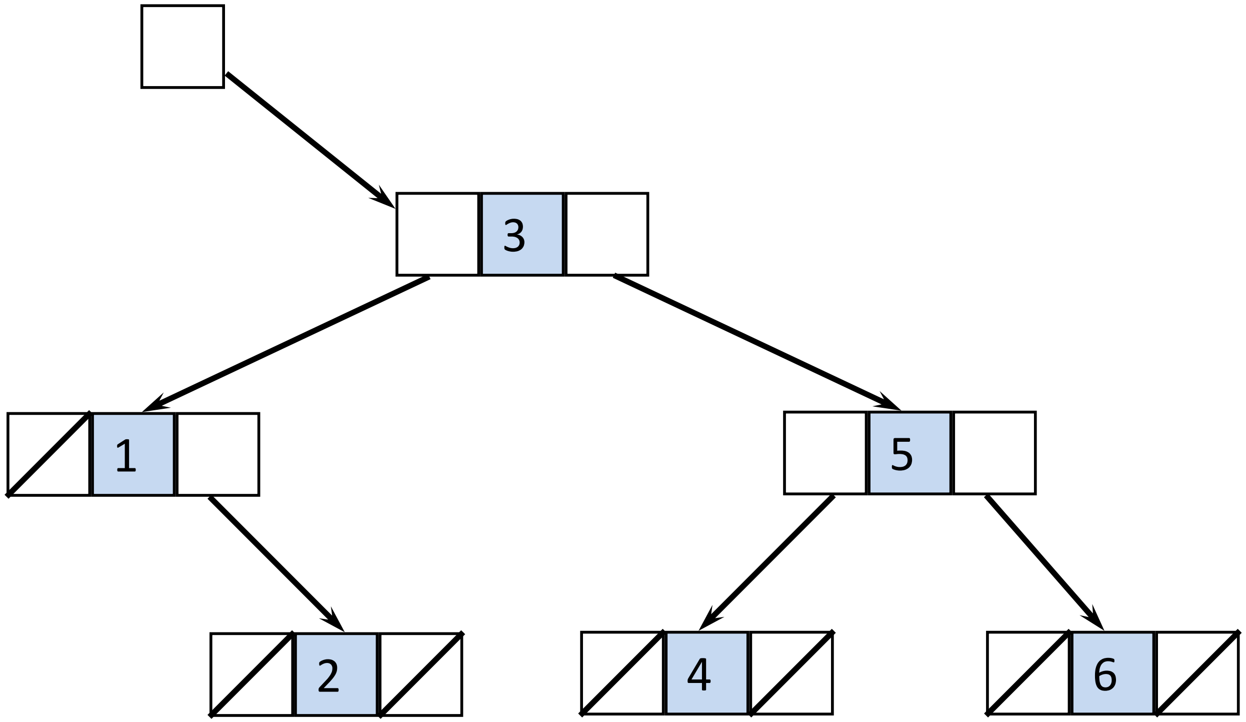
BINARY_TREE Implementation



BINARY_TREE Implementation



BINARY_TREE Implementation



BINARY_TREE Implementation