COMP9024: Data Structures and Algorithms

Trees and Binary Search Trees

Contents

- Trees
- Tree traversals
- Binary search trees

Maps and Dictionaries

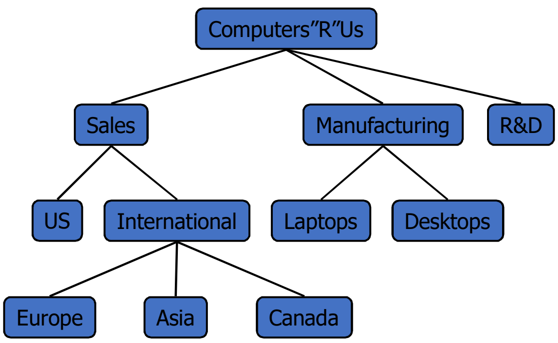
- A map (dictionary) is a collection of data items each of which is a pair (key, value), and supports the following major operations:
  - find(item): find item in the collection
  - insert(item): insert item into the collection
  - remove(item): remove item from the collection
- The only difference between a map and a dictionary is that in a map, all keys are distinct while in a dictionary, there may exist duplicate keys.
- For example, given a list of students, we can use a map to model it if student IDs are the keys, and we can use a dictionary to model it if students names are the keys.
- A value can be any data structure such as a student record.

Applications: Google, databases, online trading systems, ...

Key question: how to make the above three major operations fast?

Trees

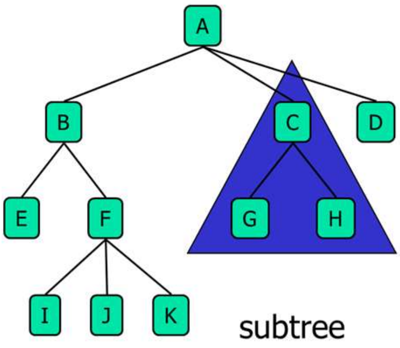
- In computer science, a tree is an abstract model of a hierarchical structure
- A tree consists of nodes with a parent-child relation
- Applications:
  - Organization charts
  - File systems
  - Programming environments



Trees Terminology

- Root: node without parent (A)
- Internal node: node with at least one child (A, B, C, F)
- Leaf node: node without children (E, I, J, K, G, H, D)
- Ancestors of a node: parent, grandparent, grand-grandparent, etc.
- Depth of a node: number of ancestors
- Height of a tree: maximum depth of any node (3)
- Descendant of a node: child, grandchild, grand-grandchild, etc.

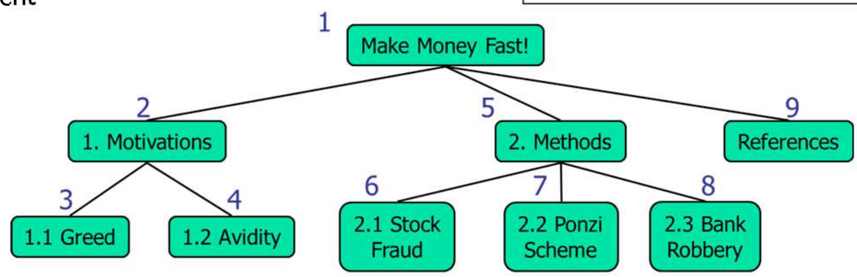
- Subtree: tree consisting of a node and its descendants



Preorder Traversal

- A traversal visits the nodes of a tree in a systematic manner
- In a preorder traversal, a node is visited before its descendants
- Application: print a structured document

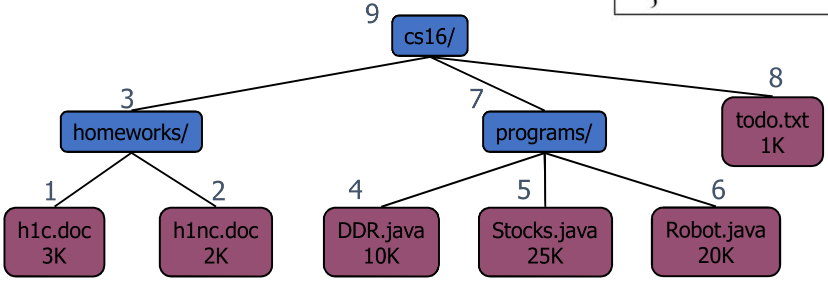
Algorithm ***preOrder***(*v*)
{ ***visit***(*v*);
for each child ***w*** of ***v***
preorder(*w*);
}



Postorder Traversal

- In a postorder traversal, a node is visited after its descendants
- Application: compute space used by files in a directory and its subdirectories

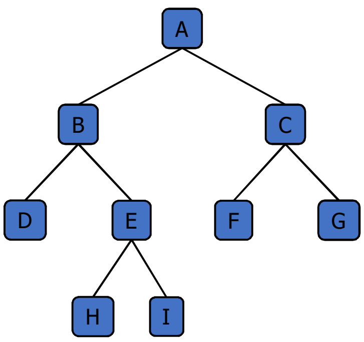
Algorithm *postOrder*(v)
{ **for each** child w of v
postOrder(w);
visit(v);
}



Binary Trees

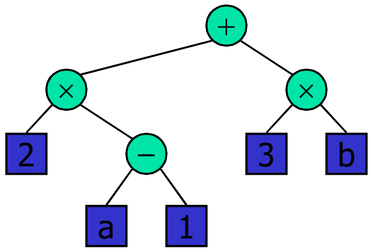
- A binary tree is a tree with the following properties:
  - Each internal node has at most two children
  - A leaf node has no child
  - The children of a node are an ordered pair
- We call the children of an internal node left child and right child
- Alternative recursive definition: a binary tree is either
  - a tree consisting of a single node, or
  - a tree whose root has an ordered pair of children, each of which is a binary tree

- Applications:
  - arithmetic expressions
  - decision processes
  - searching



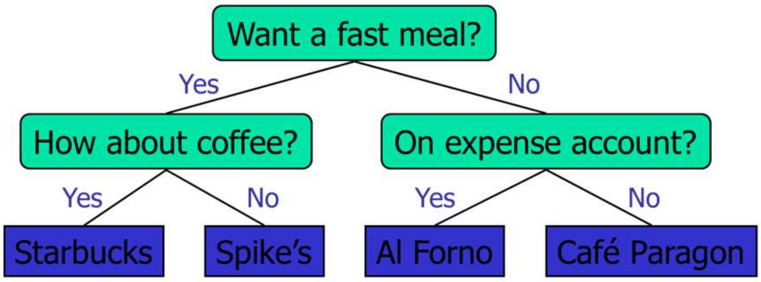
Arithmetic Expression Tree

- Binary tree associated with an arithmetic expression
  - internal nodes: operators
  - external nodes: operands
- Example: arithmetic expression tree for the expression $(2 \times (a - 1) + (3 \times b))$



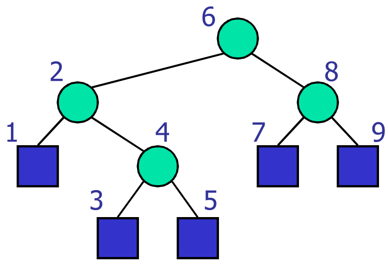
Decision Trees

- Binary tree associated with a decision process
  - internal nodes: questions with yes/no answer
  - external nodes: decisions
- Example: dining decision



Inorder Traversal

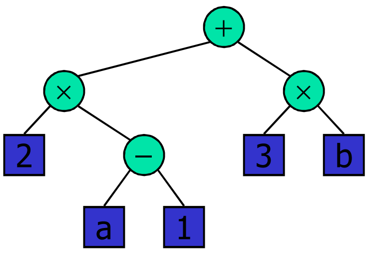
- In an inorder traversal a node is visited after its left subtree and before its right subtree
- Application: draw a binary tree
  - x(v) = inorder rank of v
  - y(v) = depth of v




```
Algorithm inOrder(v)
 { if  v as a left child
     inOrder(v.left);
   visit(v);
   if v has a right child
     inOrder(v.right);
 }
```


Print Arithmetic Expressions

- Specialization of an inorder traversal
  - print operand or operator when visiting node
  - print "(" before traversing left subtree
  - print ")" after traversing right subtree

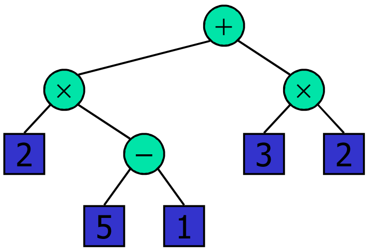



```
Algorithm printExpression(v)
  { if v has a left child
      { print("(");
        printExpression(left(v)); }
    print(v.element ());
    if  v has a Right child
      { printExpression(right(v));
        print (")"); }
  }
```

$((2 \times (a - 1)) + (3 \times b))$

Evaluate Arithmetic Expressions

- Specialization of a postorder traversal
  - recursive method returning the value of a subtree
  - when visiting an internal node, combine the values of the subtrees

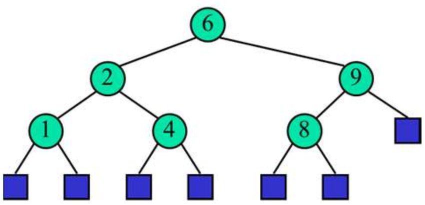


Algorithm *evalExpr*(v)
  { **if** *v is a leaf node*
    return ***v.element***;
  else
    { x = *evalExpr*(*leftChild* (v));
     y = *evalExpr*(*rightChild* (v));
     ◊ = operator stored at v;
    return x ◊ y;
    }
  }

Binary search trees

- A binary search tree is a binary tree storing keys (or key-value entries) at its nodes and satisfying the following property:
  - Let u, v, and w be three nodes such that u is in the left subtree of v and w is in the right subtree of v. We have $key(u) \le key(v) \le key(w)$
- External nodes (square nodes) denote empty nodes, and they do not store items

- An inorder traversal of a binary search trees visits the keys in non-decreasing order



External Nodes

- Advantage
  - Easier to do insertion
    - Find the place of insertion. It will be an external node.
    - Replace the external node with an internal node (and 2 external nodes)
- Disadvantage
  - Extra O(n) space (a little more than double the original!)

Binary Search Tree Operations

Operations on BSTs:

- insert(Tree,Item) ... add a new item to tree via key
- delete(Tree,Key) ... remove item with specified key from tree
- search(Tree,Key) ... find item containing key in tree
- plus, "bookkeeping" ... new(), free(), show(), ...

Representing BSTs in C

Node structure:

```
typedef struct Node {
  int  key;  // we ignore value here
  Node *left, *right;
} Node;
```

17

Searching in BSTs

TreeSearch(v, k)

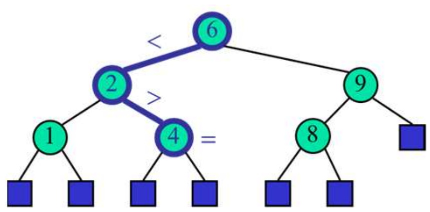
Input v (node), k (key)

Output the node containing k or null

```
if v is null or v.key=k
   return v
if k < v.key
   return TreeSearch(v.left, k)  // search left subtree
else
   return TreeSearch(v.right, k) // search right subtree
```

Time complexity: O(n)

Example: TreeSearch(root, 4)



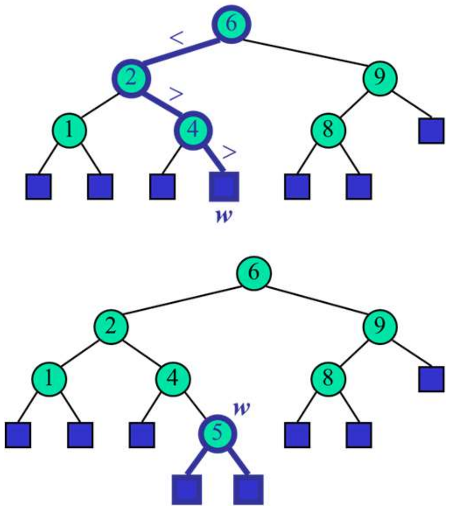
18

Insertion in BSTs (1/2)

- To insert a new item with key k, we search for key k.
- Let w be the node reached by the search
- Consider two cases:
  - k<w.key: Insert the new item as the left child of w
  - K>w.key: Insert the new item as the right child of w

Time complexity: O(n)

Example: insert 5



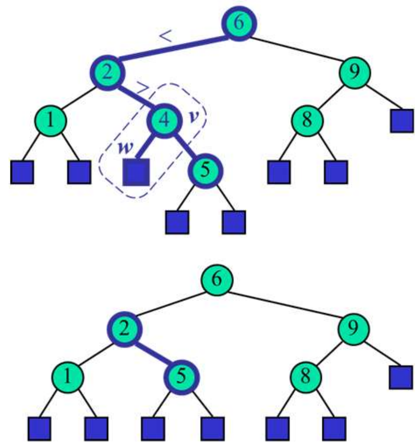
Insertion in BSTs (2/2)

```
TreeInsert(v, k)
Input  v (node), k (key)
Output: none

if root= null // empty tree
   root=CreateNode(k); // create a new node (root) to store k
else if k <= v.key // insert duplicate keys in left subtree
         {  if v.left=null
               v.left=CreateNode(k); // insert a new node as the left child of v
            else
               TreeInsert(v.left, k);
         }
      else if k> v.key
              {  if v.right=null
                    v.right=CreateNode(k); // insert a new node as the right child of v
                 else
                    TreeInsert(v.right, k);
              }
```

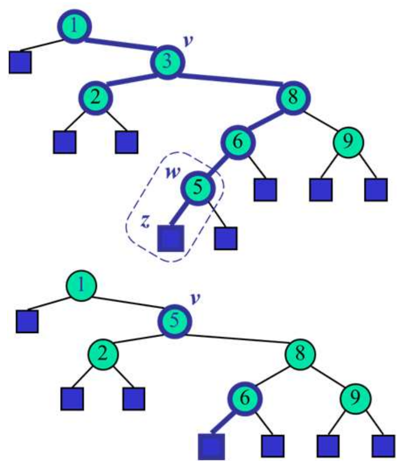
Deletion in BSTs (1/3)

- To perform operation delete(k), we search for key k
- Assume key k is in the tree, and let let v be the node storing k
- If node v has a leaf child w, we remove v and w from the tree
- Example: delete 4



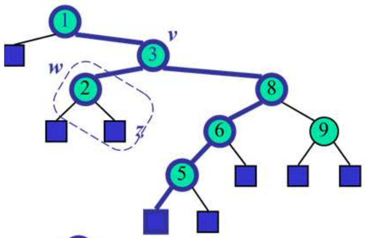
Deletion in BSTs (2/3)

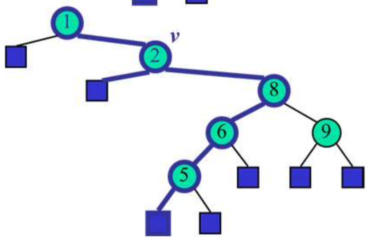
- We consider the case where the key k to be removed is stored at a node v whose children are both internal
  - We find the internal node w that follows v in an inorder traversal.
    - w is called the immediate inorder successor of v.
  - We copy $key(w)$ into node v
  - We remove node w and its left child z (which must be a leaf)
- Example: delete 3



Deletion in BSTs (3/3)

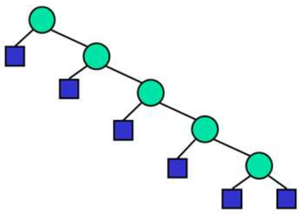
- Alternatively, we find the internal node w that precedes v in an inorder traversal.
  - w is called the immediate inorder predecessor of v.
- We copy $key(w)$ into node v
- We remove node w and its right child z (which must be a leaf)
- Example: delete 3

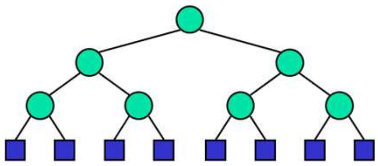




Performance

- Consider a set of n entries stored in a binary search tree of height h
  - the space used is $O(n)$
  - methods search, insert and delete take $O(h)$ time
- The height h is $O(n)$ in the worst case and $O(\log n)$ in the best case





Summary

- Trees
- Tree traversals
- Binary search trees (BSTs)
- BST search, insertion and deletion

Sedgewick, Ch.13.1-13.4