Object-Oriented Programming and Data Structures

COMP2012: Trees, Binary Trees, and Binary Search Trees

Cecia Chan
Brian Mak
Dimitris Papadopoulos

Department of Computer Science & Engineering
The Hong Kong University of Science and Technology
Hong Kong SAR, China

Review of Complexity of Algorithms: Big O Notation

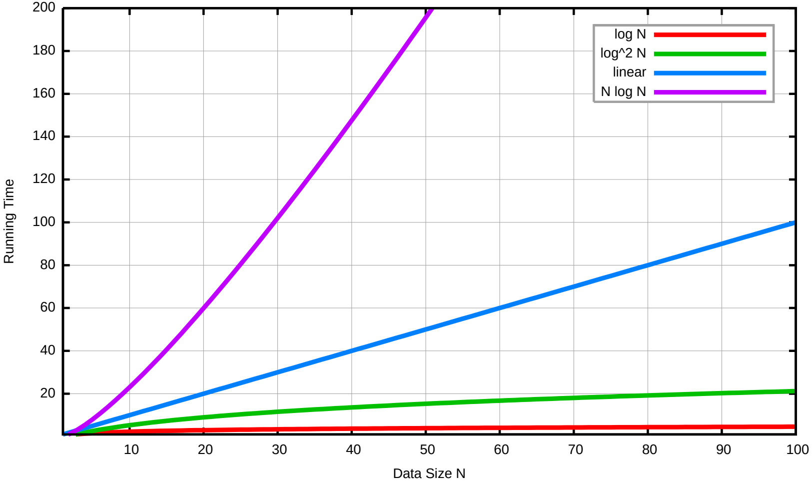
- Analyzing an algorithm allows us to predict the resources that it requires. Resources include
  - memory
  - communication bandwidth
  - computational time (usually most important)
- While the content of an input may affect the running time of an algorithm, typically, it is the input size (number of items in the input) that is the main consideration.
  - sorting: the number of items to be sorted.
  - matrix multiplication: the total number of elements in the two matrices.
- The big O notation: $O(f(N))$ gives the asymptotic upper bound of how the running time of an algorithm grows with the input size N.

Typical Growth Rates of Algorithm Running Time

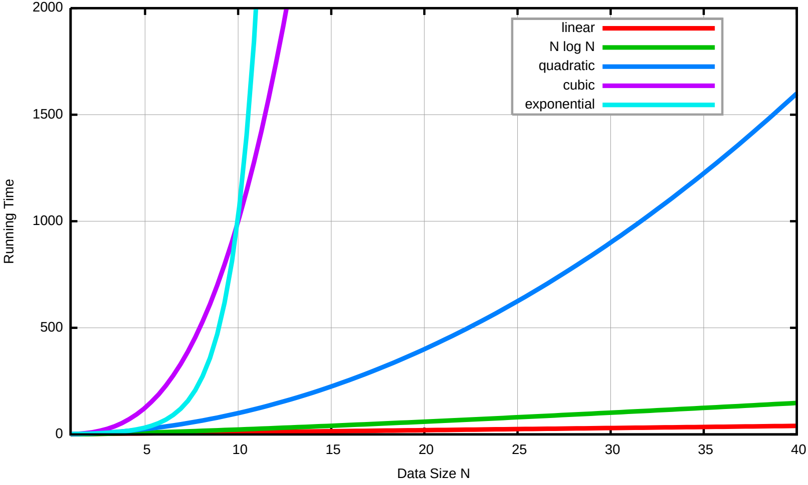
Running time $= \alpha f(N) + \beta$

Function f	**Name**
c	constant
$\log N$	logarithmic
$\log^2 N$	log-squared
N	linear
$N \log N$	
N^2	quadratic
N^3	cubic
2^N	exponential

Typical Growth Rates of Algorithm Running Time ..



Typical Growth Rates of Algorithm Running Time ...

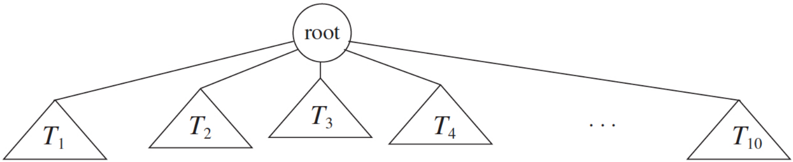


Part I

Tree Data Structure

- The linear access time $O(N)$ of linked lists is prohibitive for large amount of data.
- Does there exist any simple data structure for which the average running time of most operations (search, insert, delete) is $O(logN)$?
- Solution: Trees!
- We are going to talk about
  - basic concepts of trees
  - tree traversal
  - (general) binary trees
  - binary search trees (BST)
  - balanced trees (AVL tree)

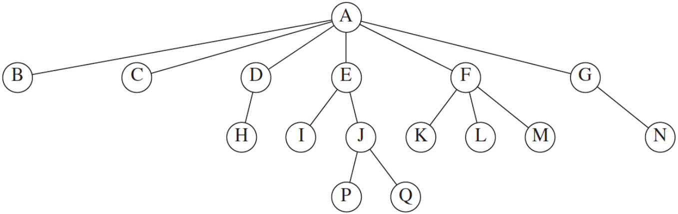
Recursive Definition of Trees



A tree T is a collection of nodes connected by edges.

- base case: T is empty
- recursive definition: If not empty, a tree T consists of
  - a root node r, and
  - zero or more non-empty sub-trees: $T_1, T_2, \ldots, T_k$

Tree Terminologies

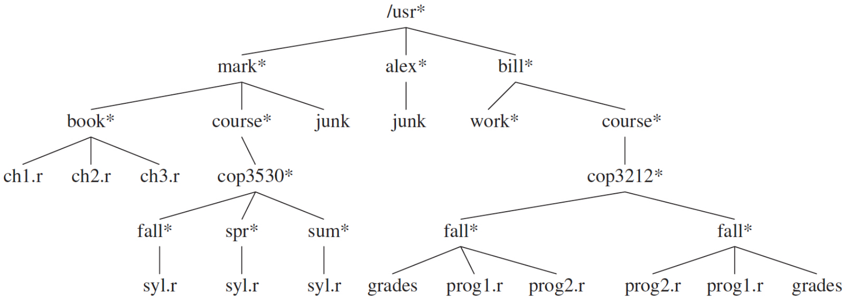


- Root: the only node with no parents
- Parent and child
  - every node except the root has exactly only 1 parent
  - a node can have zero or more children
- Leaves: nodes with no children
- Siblings: nodes with the same parent
- Path from node n_1 to n_k: a sequence of nodes $\{n_1, n_2, \ldots, n_k\}$ such that n_i is the parent of n_{i+1} for $1 \leq i \leq k-1$.

Tree Terminologies ..

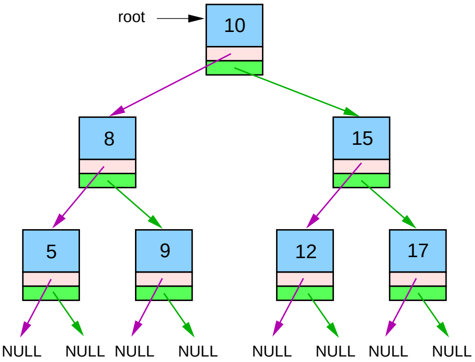
- Length of a path
  – number of edges on the path
- Depth of a node
  – length of the unique path from the root to that node
- Height of a node
  – length of the longest path from that node to a leaf
  – all leaves are at height 0
- Height of a tree
  = height of the root
  = depth of the deepest leaf
- Ancestor and descendant: If there is a path from n_1 to n_2
  – $n1$ is an ancestor of $n2$
  – $n2$ is a descendant of $n1$
  – if $n1 \neq n2$, proper ancestor and proper descendant

Example 1: Unix Directories in a Tree Structure

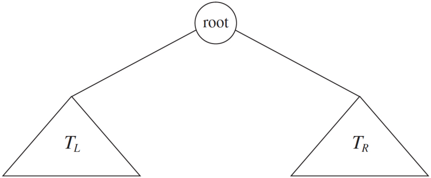


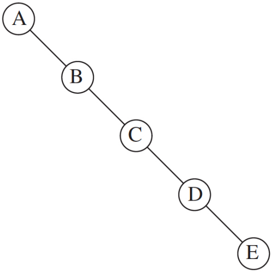
Part II

Binary Tree



Binary Trees





- Generic binary tree: A tree in which no node can have more than two children.
- The height of an 'average' binary tree with N nodes is considerably smaller than N.
- In the best case, a well-balanced tree has a height of $O(\log N)$.
- But, in the worst case, the height can be as large as $(N - 1)$.

A Typical Implementation of Binary Tree

```
#include <iostream>       /* File: btree.h */
using namespace std;

template <class T> class BTnode
{
  public:
    BTnode(const T& x, BTnode* L = nullptr, BTnode* R = nullptr)
      : data(x), left(L), right(R) { }

    ~BTnode()
    {
        delete left;
        delete right;
        cout << "delete the node with data = " << data << endl;
    }

    const T& get_data() const { return data; }
    BTnode* get_left()  const { return left; }
    BTnode* get_right() const { return right; }

  private:
    T data;               // Stored information
    BTnode* left;         // Left child
    BTnode* right;        // Right child
};
```

Binary Tree: Inorder Traversal

```
/* File: btree-inorder.cpp
 *
 * To traverse a binary tree in the order of:
 *    left subtree, node, right subtree
 * and do some action on each node during the traversal.
 */

template <class T>
void btree_inorder(BTnode<T>* root,
     void (*action)(BTnode<T>* r)) // Expect a function on r->data
{
    if (root)
    {
        btree_inorder(root->get_left(), action);
        action(root);                 // e.g. print out root->data
        btree_inorder(root->get_right(), action);
    }
}
```

Binary Tree: Preorder Traversal

```
/* File: btree-preorder.cpp
 *
 * To traverse a binary tree in the order of:
 *    node, left subtree, right subtree
 * and do some action on each node during the traversal.
 */

template <class T>
void btree_preorder(BTnode<T>* root,
     void (*action)(BTnode<T>* r)) // Expect a function on r->data
{
    if (root)
    {
        action(root);                 // e.g. print out root->data
        btree_preorder(root->get_left(), action);
        btree_preorder(root->get_right(), action);
    }
}
```

Binary Tree: Postorder Traversal

```
/* File: btree-postorder.cpp
 *
 * To traverse a binary tree in the order of:
 *    left subtree, right subtree, node
 * and do some action on each node during the traversal.
 */

template <class T>
void btree_postorder(BTnode<T>* root,
     void (*action)(BTnode<T>* r)) // Expect a function on r->data
{
    if (root)
    {
        btree_postorder(root->get_left(), action);
        btree_postorder(root->get_right(), action);
        action(root);                 // e.g. print out root->data
    }
}
```

Example 2: Binary Tree Creation & Traversal

```cpp
#include "btree.h"       /* File: test-btree.cpp */
#include "btree-preorder.cpp"
#include "btree-inorder.cpp"
#include "btree-postorder.cpp"

template <typename T>
void print(BTnode<T>* root) { cout << root->get_data() << endl; }

int main() // Build the tree from bottom up
{   // Create the left subtree
    BTnode<int>* node5 = new BTnode<int>(5);
    BTnode<int>* node9 = new BTnode<int>(9);
    BTnode<int>* node8 = new BTnode<int>(8, node5, node9);
    // Create the right subtree
    BTnode<int>* node12 = new BTnode<int>(12);
    BTnode<int>* node17 = new BTnode<int>(17);
    BTnode<int>* node15 = new BTnode<int>(15, node12, node17);
    // Create the root node
    BTnode<int>* root = new BTnode<int>(10, node8, node15);

    cout << "\nInorder traversal result:\n"; btree_inorder(root, print);
    cout << "\nPreorder traversal result:\n"; btree_preorder(root, print);
    cout << "\nPostorder traversal result:\n"; btree_postorder(root, print);
    cout << "\nDeleting the binary tree ...\n"; delete root; return 0;
}
```

Example 3: Unix Directory Traversal

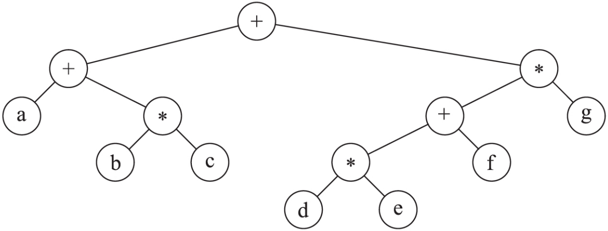
Preorder Traversal

```
/usr
    mark
        book
            ch1.r
            ch2.r
            ch3.r
        course
            cop3530
                fall
                    syl.r
                spr
                    syl.r
                sum
                    syl.r
        junk
    alex
        junk
    bill
        work
        course
            cop3212
                fall
                    grades
                    prog1.r
                    prog2.r
                fall
                    prog2.r
                    prog1.r
                    grades
```

Postorder Traversal

```
                    ch1.r
                    ch2.r
                    ch3.r
                book
                            syl.r
                        fall
                            syl.r
                        spr
                            syl.r
                        sum
                    cop3530
                course
                junk
            mark
                junk
            alex
                work
                            grades
                            prog1.r
                            prog2.r
                        fall
                            prog2.r
                            prog1.r
                            grades
                        fall
                    cop3212
                course
            bill
        /usr
```

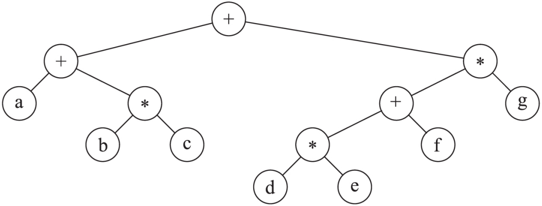
Example 4: Expression (Binary) Trees



- Above is the tree representation of the expression:

$$(a + b * c) + ((d * e + f) * g)$$

- Leaves are operands (constants or variables).
- Internal nodes are operators.
- The operators must be either unary or binary.



- Preorder traversal: node, left sub-tree, right sub-tree.

  $\Rightarrow$ Prefix notation: $+ + a * bc * + * defg$

- Inorder traversal: left sub-tree, node, right sub-tree.

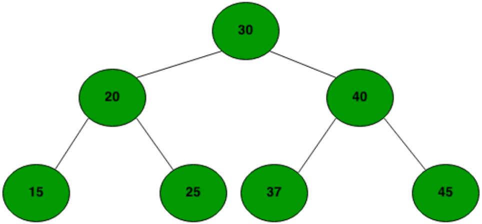
  $\Rightarrow$ Infix notation: $a + b * c + d * e + f * g$

- Postorder traversal: left sub-tree, right sub-tree, node.

  $\Rightarrow$ Postfix notation: $abc * +de * f + g * +$

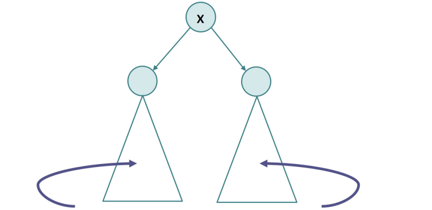
Part III

Binary Search Tree (BST)



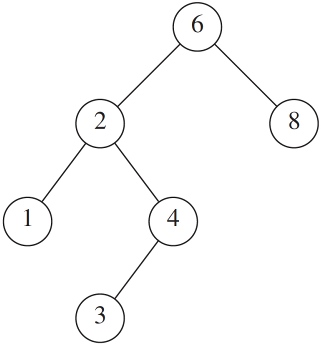
Properties of a Binary Search Tree

- BST is a data structure for efficient searching, insertion and deletion.
- BST property: For every node x
- All the keys in its left sub-tree are smaller than the key value in node x.
- All the keys in its right sub-tree are larger than the key value in node x.

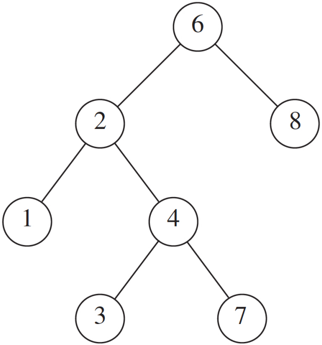


BST Example and Counter-Example

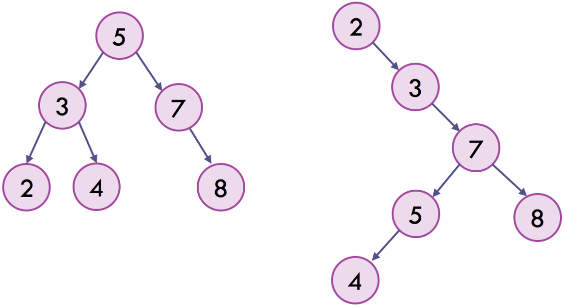
BST



Not a BST but a Binary Tree

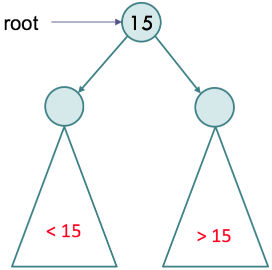


BSTs May Not be Unique



- The same set of values may be stored in different BSTs.
- Average depth of a node on a BST is O(log N).
- Maximum depth of a node on a BST is O(N).

BST Search

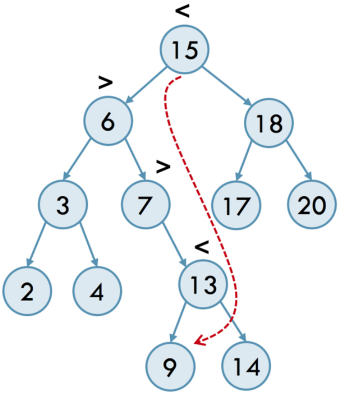


For the above BST, if we search for a value

- of 15, we are done at the root.
- $<$ 15, we would recursively search it with the left sub-tree.
- $>$ 15, we would recursively search it with the right sub-tree.

Example 5: BST Search

- Let's search for the value 9 in the following BST.



Compare	Action
9 vs. 15	continue with the left subtree
9 vs. 6	continue with the right subtree
9 vs. 7	continue with the right subtree
9 vs. 13	continue with the left subtree
9 vs. 9	eureka!

Our BST Implementation (different from textbook's)

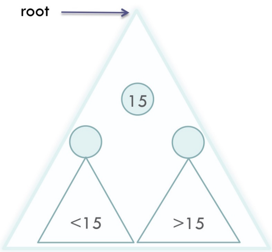
```
template <typename T> class BST /* File: bst.h */
{
  private:
    struct BSTnode      // A node in a binary search tree
    {
        T value;
        BST left;        // Left sub-tree or called left child
        BST right;       // Right sub-tree or called right child
        BSTnode(const T& x) : value(x) { } // Assume a copy cons. for T
        // same as: BSTnode(const T& x) : value(x), left(), right() { }
        BSTnode(const BSTnode&) = default; // Copy constructor
        ~BSTnode() { cout << "delete: " << value << endl; }
    };

    BSTnode* root = nullptr;

  public:
    BST() = default;              // Empty BST
    ~BST() { delete root; }     // Actually recursive
```

Our BST Implementation (different from textbook's) ..

```
    // Shallow BST copy using move constructor
    BST(BST&& bst) { root = bst.root; bst.root = nullptr; }

    BST(const BST& bst)  // Deep copy using copy constructor
    {
        if (bst.is_empty())
            return;

        root = new BSTnode(*bst.root);  // Recursive
    }

    bool is_empty() const { return root == nullptr; }
    bool contains(const T& x) const;
    void print(int depth = 0) const;
    const T& find_max() const; // Find the maximum value
    const T& find_min() const; // Find the minimum value

    void insert(const T&);      // Insert an item with a policy
    void remove(const T&);      // Remove an item
};
```

Our BST Implementation

- Our implementation really implements a BST as an object.
- It has a root pointing to a BST node which has
  - a value (of any type)
  - a left BST object: a sub-tree with values smaller than that of the root.
  - a right BST object: a sub-tree with values greater than that of the root.



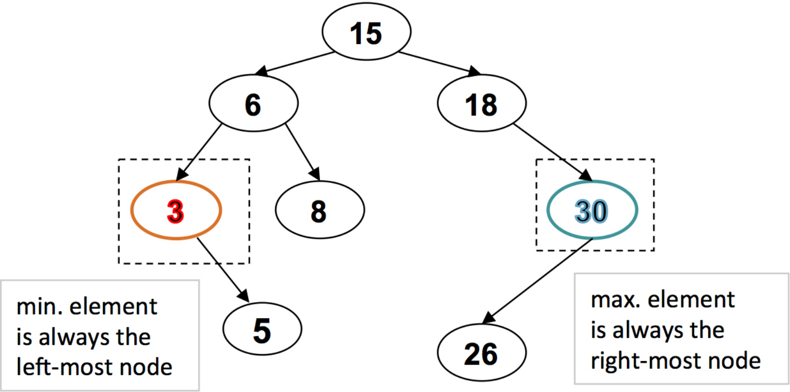
BST Code: Search

```
/* Goal: To check if the BST contains the value x.
 * Return: (bool) true or false
 * Time complexity: O(height of BST)
 */

template <typename T>          /* File: bst-contains.cpp */
bool BST<T>::contains(const T& x) const
{
    if (is_empty())            // Base case #1
        return false;

    if (root->value == x)      // Base case #2
        return true;

    else if (x < root->value) // Recursion on the left sub-tree
        return root->left.contains(x);

    else                       // Recursion on the right sub-tree
        return root->right.contains(x);
}
```

BST Code: Print by Rotating it -90 Degrees

```
/* Goal: To print a BST
 * Remark: The output is the BST rotated -90 degrees
 */

template <typename T>                   /* File: bst-print.cpp */
void BST<T>::print(int depth) const
{
    if (is_empty())                     // Base case
       return;

    root->right.print(depth+1);         // Recursion: right sub-tree

    for (int j = 0; j < depth; j++) // Print the node value
        cout << '\t';
    cout << root->value << endl;

    root->left.print(depth+1);          // Recursion: left sub-tree
}
```

BST: Find the Minimum/Maximum Stored Value



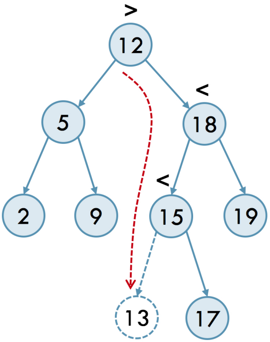
BST Code: Find the Minimum Stored Value

```
/* Goal: To find the min value stored in a non-empty BST.
 * Return: The min value
 * Remark: The min value is stored in the leftmost node.
 * Time complexity: O(height of BST)
 */

template <typename T>   /* File: bst-find-min.cpp */
const T& BST<T>::find_min() const
{
    const BSTnode* node = root;

    while (!node->left.is_empty()) // Look for the leftmost node
        node = node->left.root;

    return node->value;
}
```

BST Code: Find the Maximum Stored Value

```
/* Goal: To find the max value stored in a non-empty BST.
 * Return: The max value
 * Remark: The max value is stored in the rightmost node.
 * Time complexity: O(height of BST)
 */

template <typename T>   /* File: bst-find-max.cpp */
const T& BST<T>::find_max() const
{
    const BSTnode* node = root;

    while (!node->right.is_empty()) // Look for the rightmost node
        node = node->right.root;

    return node->value;
}
```

BST: Insert a Node of Value x

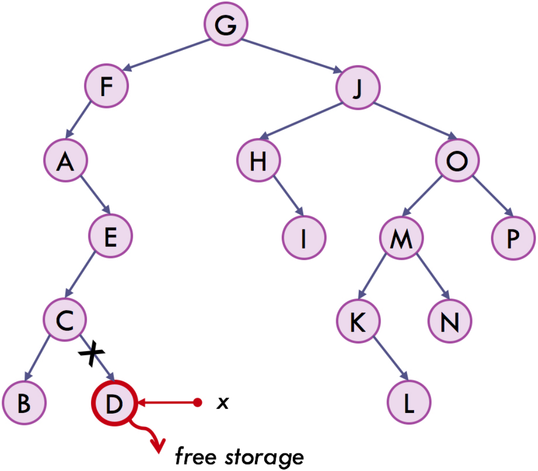


- E.g., insert 13 to the BST.
- Proceed down the tree as you would with a search.
- If x is found, do nothing (or update something).
- Otherwise, insert x at the last spot on the path traversed.
- Time complexity = O(height of the tree)

BST Code: Insertion

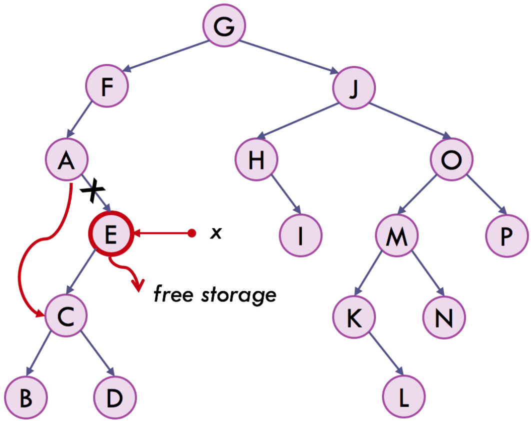
```
template <typename T>          /* File: bst-insert.cpp */
void BST<T>::insert(const T& x)
{
    if (is_empty())            // Find the spot
        root = new BSTnode(x);

    else if (x < root->value)
        root->left.insert(x);  // Recursion on the left sub-tree

    else if (x > root->value)
        root->right.insert(x); // Recursion on the right sub-tree

    else // This line is optional; just for illustration
        ;                      // x == root->value; do nothing
}
```

BST: Delete a Leaf



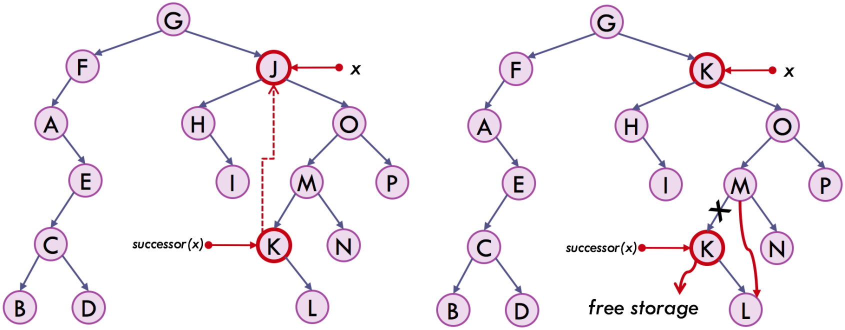
- Delete the leaf node immediately.

BST: Delete a Node with 1 Child



- Adjust a pointer from its parent to bypass the deleted node.

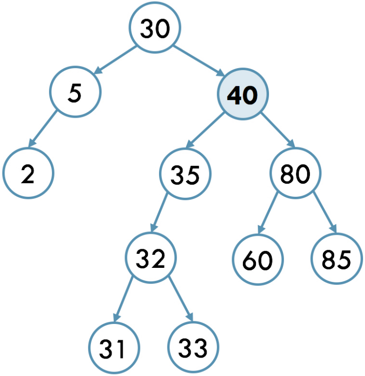
BST: Delete a Node with 2 Children



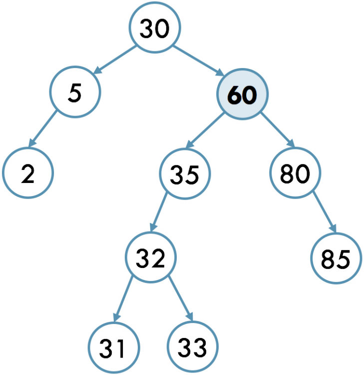
- You will have 2 choices: replace the deleted node with the
  - maximum node in its left sub-tree, or
  - minimum node in its right sub-tree (as in the above figure).
- Remove the max/min node depending on the choice above.

Example 6.1: BST Deletions

- Removing 40 from BST(a), replacing it with the min. value in its right sub-tree results in the BST(b).



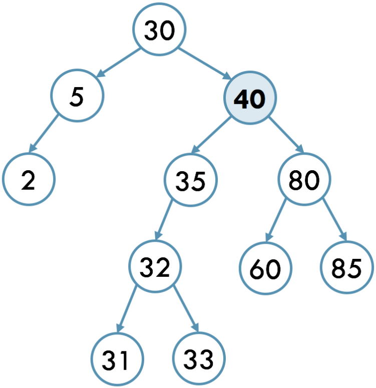
(a)



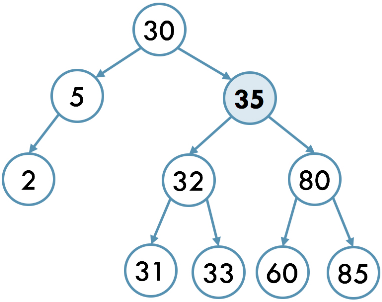
(b)

Example 6.2: BST Deletions

- Removing 40 from BST(a), replacing it with the max. value in its left sub-tree results in the BST(c).

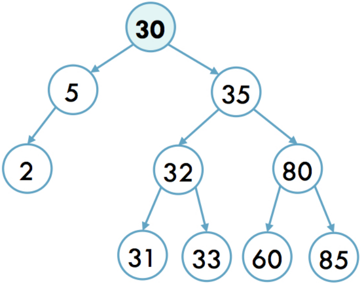


(a)

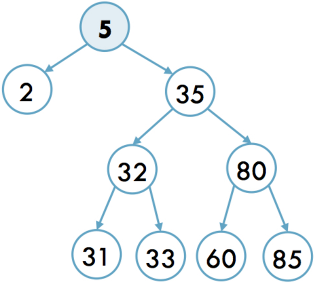


(c)

- Removing 30 from BST(c) and moving 5 from its left sub-tree result in BST(d).



(c)



(d)

BST Code: Deletion

```
template <typename T>           /* File: bst-remove.cpp */
void BST<T>::remove(const T& x) // leftmost item of its right subtree
{
    if (is_empty())             // Item is not found; do nothing
        return;

    if (x < root->value)        // Remove from the left subtree
        root->left.remove(x);
    else if (x > root->value)   // Remove from the right subtree
        root->right.remove(x);
    else if (root->left.root && root->right.root) // Found node has 2 children
    {
        root->value = root->right.find_min(); // operator= defined?
        root->right.remove(root->value); // min is copied; can be deleted now
    }
    else                        // Found node has 0 or 1 child
    {
        BSTnode* deleting_node = root; // Save the root to delete first
        root = (root->left.is_empty()) ? root->right.root : root->left.root;

        // Set subtrees to nullptr before removal due to recursive destructor
        deleting_node->left.root = deleting_node->right.root = nullptr;
        delete deleting_node;
    }
}
```

```
#include <iostream>      /* File: test-bst.cpp */
using namespace std;
#include "bst.h"
#include "bst-contains.cpp"
#include "bst-print.cpp"
#include "bst-find-max.cpp"
#include "bst-find-min.cpp"
#include "bst-insert.cpp"
#include "bst-remove.cpp"

int main()
{
    BST<int> bst;
    while (true)
    {
        char choice; int value;
        cout << "Action: d/f/i/m/M/p/q/r/s "
             << "(deep-cp/find/insert/min/Max/print/quit/remove/shallow-cp): ";
        cin >> choice;

        switch (choice)
        {
```

BST Testing Code ..

```
case 'd': // Deep copy
{
    BST<int>* bst2 = new BST<int>(bst);
    bst2->print(); delete bst2;
}
    break;
case 'f': // Find a value
    cout << "Value to find: "; cin >> value;
    cout << boolalpha << bst.contains(value) << endl;
    break;
case 'i': // Insert a value
    cout << "Value to insert: "; cin >> value;
    bst.insert(value);
    break;
case 'm': // Find the minimum value
    if (bst.is_empty())
        cerr << "Can't search an empty tree!" << endl;
    else
        cout << bst.find_min() << endl;
    break;
case 'M': // Find the maximum value
    if (bst.is_empty())
        cerr << "Can't search an empty tree!" << endl;
```

BST Testing Code ..

```
                else
                    cout << bst.find_max() << endl;
                break;
            case 'p': // Print the whole tree
            default:
                cout << endl; bst.print();
                break;
            case 'q': // Quit
                return 0;
            case 'r':
                cout << "Value to remove: "; cin >> value;
                bst.remove(value);
                break;
            case 's': // Shallow copy
            {
                BST<int> bst3 { std::move(bst) };
                bst3.print();
                bst.print();
            }
                break;
        }
    }
}
```