BINARY TREE EXAMPLE 1: ONE TEMPLATE VARIABLE

Managing orderable objects

Delroy A. Brinkerhoff

ASSOCIATIVE DATA STRUCTURES AND SEARCHES

Name	Address	
Dilbert	225 Elm	…
Alice	256 N 400 W	…
Wally	718 Washington	…
Asok	633 Adams	…

- Associative data are a set of related values
- Implemented as objects
- Viewed as a table
  - Rows correspond to objects
  - Columns correspond to member variables
- An object is accessed by a key, making the associated values available

THE Employee CLASS

```
class Employee
{
    private:
        string name;
        string address;

    public:
        Employee(string n = "", string a = "")
            : name(n), address(a) {}

        bool operator==(Employee& e) { return name == e.name; }
        bool operator<(Employee& e) { return name < e.name; }

        friend ostream& operator<<(ostream& out, Employee& me)
        {
            out << me.name << " " << me.address;
            return out;
        }
};
```

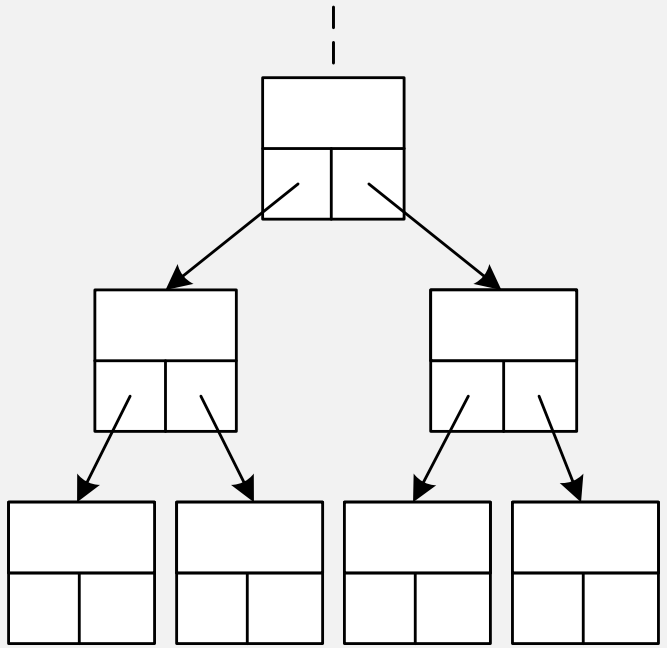
THE Tree CLASS

```
template <class T>
class Tree
{
    private:
        T          data;
        Tree<T>*   left = nullptr;
        Tree<T>*   right = nullptr;
    public:
        ~Tree();
        T* insert(T key);
        T* search(T key);
        void remove(T key);
};
```

```
#include <iostream>
#include <string>
#include "Tree.h"
#include "Employee.h"
using namespace std;


int main()
{
    Tree<Employee>  tree;
        . . .
}
```

RECURSIVE DATA STRUCTURES

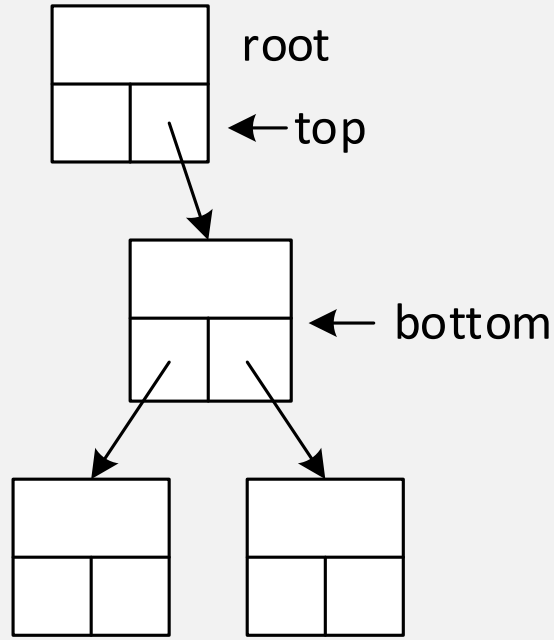


```
template <class T>
Tree<T>::~Tree()
{
    if (left != nullptr)
        delete left;
    if (right != nullptr)
        delete right;
    //cout << data << endl;
}
```

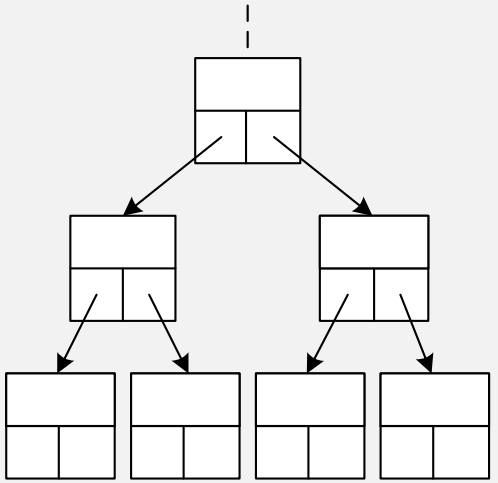
POINTER INITIALIZATION

- `Tree<T>* top = this;`
- `Tree<T>* bottom = right;`

- `top = bottom;`
- `Tree<T>* succ = bottom->right;`



DESCENDING THE TREE: SELECTING A SUBTREE



- `top = bottom;`
- ```
  if (key < bottom->data)
      bottom = bottom->left;
  else
      bottom = bottom->right;
  ```
- `bottom = (key < bottom->data) ? bottom->left : bottom->right;`
- `((top != this && key < top->data) ? top->left : top->right) = bottom;`
  ```

THE Tree insert FUNCTION

```
template <class T>
T* Tree<T>::insert(T key)
{
    Tree<T>* top = this;
    Tree<T>* bottom = right;

    while (bottom != nullptr)
    {
        if (bottom->data == key)
            return &bottom->data;

        top = bottom;
        bottom = (key < bottom->data) ? bottom->left : bottom->right;
    }

    bottom = new Tree;
    bottom->data = key;

    ((top != this && key < top->data) ? top->left : top->right) = bottom;

    return &bottom->data;
}
```

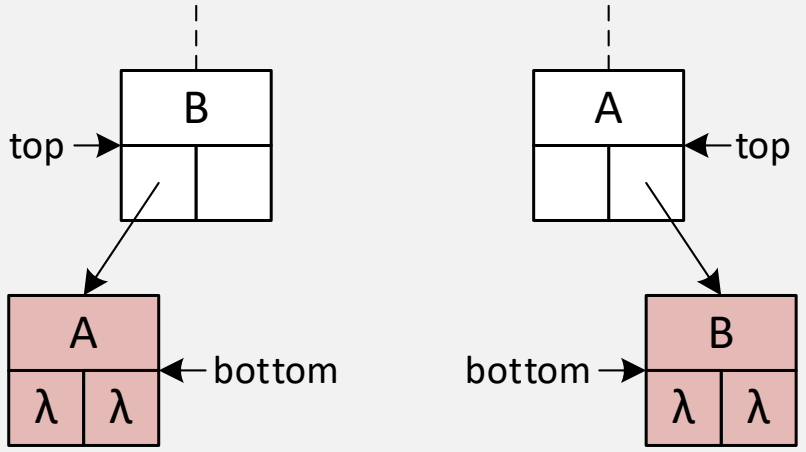
REMOVING TREE NODES (1)

```
template <class T>
int Tree<T>::subtrees()
{
    if (left == nullptr && right == nullptr)
        return 0;
    else if (left == nullptr || right == nullptr)
        return 1;
    else
        return 2;
}
```

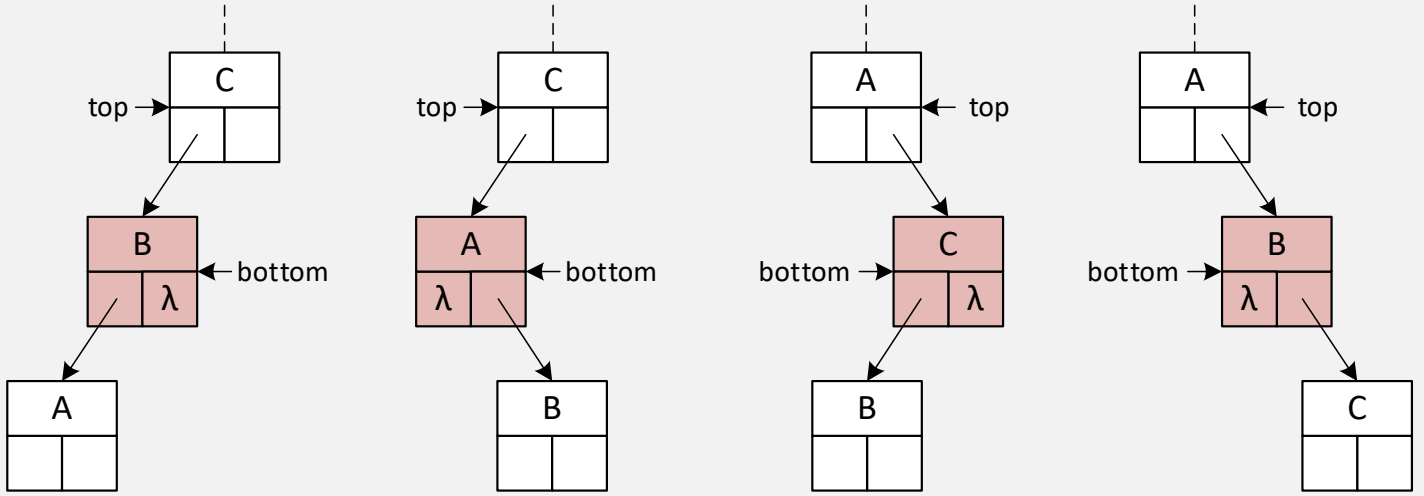
```
template <class T>
void Tree<T>::remove(Tree<T>* top, Tree<T>* bottom)
{
    switch (bottom->subtrees())
    {
        case 0:
            . . .
        case 1:
            . . .
        case 2:
            . . .
    }
}
```

REMOVING TREE NODES (2)

```
case 0:
    //cout << "CASE 1" << endl;
    if (top->left == bottom)
        top->left = nullptr;
    else
        top->right = nullptr;
    delete bottom;
    return;
```



REMOVING TREE NODES (3)




```
case 1:
    //cout << "CASE 2" << endl;
    if (top->left == bottom)
        top->left = (bottom->right == nullptr) ? bottom->left : bottom->right;
    else if (top->right == bottom)
        top->right = (bottom->right == nullptr) ? bottom->left : bottom->right;

    bottom->left = bottom->right = nullptr;
    delete bottom;
    return;
```

REMOVING TREE NODES (4)

```
case 2:
    //cout << "CASE 3" << endl;
    top = bottom;
    Tree<T>* succ = bottom->right;
    while (succ->left != nullptr)
    {
        top = succ;
        succ = succ->left;
    }
    bottom->data = succ->data;
    remove(top, succ);
    return;
```