Lec_28.md

Lecture 28

Nov. 19/2020

Binary Search Trees

TreeNode Class Definition

```
class treeNode {
private:
  int key;
  treeNode* left;
  treeNode* right;
public:
  treeNode();
  treeNode(int k);
  treeNode(int k, treeNode* l, treeNode* r);
  int getKey() const;
  treeNode* getLeft() const;
  treeNode* getRight() const;
  void setKey(int k);
  void setLeft(treeNode* l);
  void setRight(treeNode* r);
  void print() const;
};
```

TreeNode Constructors

```
treeNode::treeNode() {
key = 0;
left = NULL;
right = NULL;
};

treeNode::treeNode(int k) {
key = k;
left = NULL;
right = NULL;
};

treeNode::treeNode(int k, treeNode* l, treeNode* r) {
key = k;
left = l;
right = r;
};
```

Notes:

1. `treeNode()`

- Can define/instantiate a **BST** as empty

2. `treeNode(int k)`

- Or with a **root** key `k`

3. `treeNode(int k, treeNode* l, treeNode* r)`

- Or with a **root** node and left/right subtrees

TreeNode Accessors/Mutators

```
int treeNode::getKey() const {
  return (key);
}
treeNode* treeNode::getLeft() const {
  return (left);
}
void treeNode::setLeft(treeNode* l) {
  left = l;
}
```

Tree Class Definition

```
class Tree {
  private:
  treeNode* root;
public:
  Tree ();
  Tree (const Tree& other);
  ~Tree ();
  Tree& operator=(const Tree& rhs);
  :
  :
  void insert(treeNode* p);
};
```

Tree Constructor

```
Tree::Tree() {
  root = NULL;
};
```

Tree Destructor

Must be a deep implementation

- We need to de-allocate the tree before de-allocating the object

Deallocating the tree

- The node root should be deleted ***last***
  - To avoid memory leaks (dangling pointers)

```
Tree::~Tree(){
  if(root==NULL){
    return;
  }
  delete_tree(treeNode* root);
}
```

Notes:

1. `delete_tree(treeNode* root);`

- Why do we need this?
  - Because we cannot explicitly call `~Tree`
  - And we want to use recursion to delete the tree
- `delete_tree` is a **helper function** that facilitates recursion
- Also, because `~Tree` is a member of class `Tree`
  - We want to deallocate all the nodes of *first*

```
Tree::delete_tree (treeNode* myroot) {
  if (myroot == NULL) return;
  delete_tree(myroot->getLeft());
  delete_tree(myroot->getRight());
  delete myroot;
}
```

Notes:

1. `if (myroot == NULL) return;`

- **basis** for recursion

2. `delete_tree(myroot->getLeft());`

- Delete left subtree

3. `delete_tree(myroot->getRight());`

- Delete right subtree

4. `delete myroot;`

- Delete **root** node
- The **root** node is deleted *last*

Tree Insertion

```
void Tree::insert(treeNode* p) {
  if (p == NULL) return; // Nothing to insert
  if (root == NULL) { // basis
    root = p; root->setLeft (NULL); root->setRight(NULL);
    return;
  }
  // Helper function to facilitate the recursion
  insert_bst(p, root);
}
```

Notes:

1. `insert_bst(p,root);`

- Why do we need to use a **helper** function again?
  - Need an additional parameter for recursion (determine where to place the inseritng node)

```
void Tree::insert_bst(treeNode* p, treeNode* r) {
  if (p->getKey() == r->getKey()) return;
```

```
  if (p->getKey() < r->getKey()) {
    if (r->getLeft() == NULL) {
      r->setLeft(p);
      return;
    }
  else insert_bst(p, r->getLeft());
  }

  if (p->getKey() > r->getKey()) {
    if (r->getRight() == NULL) {
      r->setRight(p);
      return;
    }
  else insert_bst(p, r->getRight());
  }

}
```

Notes:

1. `insert_bst` can be a **private** function

- `insert_bst` serves only to facilitate recursion
  - Make this `private` to prevent incorrect access

2. `if (p->getKey() == r->getKey()) return;`

- Recursive **basis**

Copy Constructor

```
void preorder (treenode *root) {
  if (root != null) {
    cout << root->data;
    preorder (root->left);
    preorder (root->right);
  }
}
```

Inheritance in C++

Inheritance is a C++ mechanism that facilitates code reuse

- Shows up in other programming languages as well (e.g. Java, Python)

Allows programmers to extend/enhance existing classes without modifying the code in these classes

- **Open Closed Principle**
  - Open for extension, Closed for modification

Inheritance Example: Name Class

```
class Name {
  private:
    char * theName;
  public:
    Name();
    Name(constchar* name);
    Name(Name & r);
    ~Name();
    void setName(constchar* newName);
```

```
    Name & operator=(Name & r);
    void print();
};
```

Notes:

- Think of the string class we implemented a while ago
  - Base class for an **inheritance** example

Name Class Implementation

Constructors

```
Name::Name() {
  theName = new char [1];
  theName[0] = '\0';
}
Name::Name(const char* name) {
  theName = new char[strlen(name)+1];
  strcpy(theName, name);
}
Name::~Name() {
  delete [] theName;
}
```

Notes:

1. `theName[0] = '\0';`

- Null terminated strings (cstrings)

2. `strcpy(theName, name);`

- String copy (from C standard library)

Accessors/Mutators

```
void Name::setName(const char* newName) {
  delete [] theName;
  theName = new char[strlen(newName)+1];
  strcpy(theName, newName);
}
void Name::print() {
  cout << theName << endl;
}
```

Inheritance Example: Contact Class

Now, say we want to implement a contact class, which contains the following:

- Name of a contact
  - Set name
  - Get name
- Address of a contact
  - Set address
  - Get address

We could copy over the functions/data members from String

- No code reuse
- Start from scratch

Or, we could ask to pay for a license to use the original `Name` class source code

- This works, and we get original `Name` class code
  - However, imagine if the `Name` class has 1000000000 lines
    - We need to learn how it works
    - Can sometimes takes months to understand the class
- What if the original class updates to a new version with more functionality?

But what about **Inheritance**?

- Allows programmers and engineers to *extend* the capability of a class
- Reuse the code *even when source is not provided*
- Only need to understand what Name does
  - Not how it does it
    - **Complexity management**
  - Only need to understand what the public data members are (object functionality)
- Updates to `Name` *automatically* reflected in `Contact`

Contact Class Implementation

```
#include "Name.h"

class Contact: public Name{
private:
  char * theAddress;
public:
  Contact();
  ~Contact();
  Contact(Contact & r);
  Contact(const char* newName,
  const char* newAddress);
  void setNameAddress(const char* newName,
  const char* newAddress);
  Contact & operator=(Contact & r);
  void print();
};
```

Notes:

1. `class Contact: public Name`

- This line reads as:
  - class Contact inherits Name publicly
- This defines inheritance: Contact inherits members from Name (publicly)

2. All of the other Contact methods are available to use for Contact objects

- And now *all* of the `Name` methods are *available to use* for Contact objects

`Contact` inherits from `Name` all the data members

- As well as function members
  - There are a couple of exceptions

A `Contact` object looks like this:

```
theName
:
setName(...)
print()

theAddress
:
setNameAddress(..)
print()
```

Note:

- Both methods in `Name` and `Contact` accessible with `Contact` Object

Contact Class Usage

```
#include "Name.h"
#include "Contact.h"

int main() {
  Name n;
  Contact c;

  n.setName(“Tarek Abdelrahman”);

  n.print();

  c.setName(“Tarek Abdelrahman”);

  c.setNameAddress(“John Smith”, “123 Main Street”);

  n.setNameAddress(“Tarek Abdelrahman”,“123 Main Street”);

  c.print();

  return (0);
}
```

Notes:

1. `n.setName(“Tarek Abdelrahman”);`

- Can use `Name` object as normal

2. `n.print();`

- This works normally as well

3. `c.setName(“Tarek Abdelrahman”);`

- This works, since `Contact` inherits `Name`

4. `c.setNameAddress(“John Smith”, “123 Main Street”);`

- This works, since `setNameAddress` defined in `Contact` class

5. `n.setNameAddress(“Tarek Abdelrahman”,“123 Main Street”);`

- This is a **compile-time error**
  - `Name` class has no method `setNameAddress`
  - **Inheritance** works one way

6. `c.print();`

- This works, since there is a `print()` method in `Contact`; *however*,
    - `print()` is *also* defined in `Name`
    - So which `print()` gets called?
        - The general idea is that the sub gets called
        - The function in the class that is inheriting (`Contact`) gets called
- `print()` in `Contact` *eclipses* `print()` inherited from `Name`

Note:

- We can *choose* which overloaded function we want to call
    - There is a way to call `print()` from `Name`
        - By default, `print()` from `Contact` is called (as it eclipses the inherited `print()`)