Lec_22.md

Lecture 22 - An Abstract String Class and Linked Lists

Oct. 29/2020

String Class

Continuing the string class implementation from last lecture...

Comparison Operators

operator==

There are multiple cases for the definition of `operator==` . Compare:

1. `fname == "Stewart";`
2. `"Tarek" == FirstName;`
3. `fname == FirstName;`

So need to define multiple `operator==` calls

Comparing String and String

Comparing two objects of class `String` :

```
bool String::operator==(const String & rhs) const {
  return (strcmp(str, rhs.str) == 0);
}
```

Comparing String and const char*

```
bool String::operator==(const char* s) const {
  return (strcmp(str, s) == 0);
}
```

Comparing const char* and String

What if we have a string on the LHS?

- e.g. `"Stewart" == fname`
  - Build an `operator==` **overload** that is a **non-member function**

```
bool operator==(const char* lhs,const String & rhs) const {
  return (strcmp(lhs, rhs.str) == 0);
}
```

Note:

1. `bool operator==(const char* lhs,const String & rhs) const { }`

- `operator==` is a **non-member function**

2. `return (strcmp(lhs, rhs.str) == 0);`

- `operator==` is trying to access `rhs.str`
  - But this overload is a **non-member function**
    - So cannot access by default
    - Need to make `operator==` a **friend** function of class `String`

Printing String Objects

```
ostream & operator<<(ostream & os, const String & s)
{
  os << "(" << s.str << ")";
  return (os);
}
```

Notes:

1. `ostream & operator<<(ostream & os, const String & s){ }`

- `operator<<` is a **non-member function**
  - Need to make `operator<<` a **friend** function of class `String`
- Can pass `ostream & os` as a `const` if you would like :)

Overall Class Header File

```
class String {
  private:
    char *str;
    int len;
  public:
    String();
    String(const String & s);
    String(const char * s);
    ~String();
    // Member functions
    :
    :
    friend bool operator==(const char *, const String &);
    friend String operator+(const char *, const String &);
    friend ostream & operator<<(ostream &, const String &);
};
```

Notes:

1. `friend bool operator==(const char *, const String &);`

- Friend definition for `operator==` with LHS `const char *`

2. Note two other friend definitons

- `operator+`
- `operator<<`

And that concludes our `String` class definition!!

- Full code should be on quercus

Linked List

Moving on to a new subject: **Linked Lists**

- A class of data structures that are used as building blocks in many applications
- **Dynamic Structures**
  - Can shrink and grow
  - Unlimited/Unfixed size
- More efficient addition/deletion of elements

Operations on **Linked Lists**:

- **Traversal**
- Locating nodes
- Inserting nodes
- Deleting nodes

Linked List Definition

Linked Lists:

- Arbitrarily long sequence of **nodes**
  - Each **node** contains
    a. A data field (**key**)
    b. A pointer to the next **node** in the list
- **Linked List**'s have a **head** which points to the first node in the list
- If key1 < key2 < key3 < ... < keyn
  - The list is **sorted/ordered**

Linked List Class Definitions

```
class listNode{
public:
  int key;
  listNode* next;
};

listNode node;
```

1. Members are public only for presentation purposes
2. `int key;`

- `listNode` contains a key

3. `listNode* next;`

- `listNode` also contains pointer to next element in the **Linked List**

```
class linkedList{
public:
  listNode* head;
};

linkedList myList;
```

Notes:

1. Members are public only to simplify presentation

2. `listNode* head;`

- `linkedList` object contains a pointer to the *first* or **head** node of the **linked list**

Traversing a Linked List

The idea is to visit each node once and process the data

```
void linkedList::traverseList( ) {
  listNode* tptr = head;
  while (tptr != NULL) {
    cout << tptr->key << endl;
    tptr = tptr->next;
  }
}
```

Notes:

1. `listNode* tptr = head;`

- Start by setting a **Traversal Pointer** to the **head** node

2. `while(tptr!=NULL){ }`
3. `tptr = tptr->next;`

- Set the **traversal pointer** to the *next* node in the list
  - On next iteration of `while`, `tptr` refers to the *next* node in list

4. This works even if the list is empty

- Since `tptr!=NULL` will evaluate to false

Locating a Node in a Linked List

```
listNode* linkedList::LocateInList(int k) {
  listNode* tptr = head;
    while (tptr != NULL) {
    if (tptr->key == k) break;
    tptr = tptr->next;
    }
  return tptr;
}
```

Process:

1. Iterate through the list by using **traversal pointers**
2. Check if `tptr->key == k`

- Check if we have found our node
- If so, we have found our node, so `break;`

3. Set `tptr = tptr->next;`

Notes:

- This works as well if list is empty, since `tptr!=NULL` will also evaluate to false

Inserting a value at Head

```
void linkedList::insertAtHead(listNode* nptr) {
  nptr->next = head;
  head = nptr;
}
```

Notes:

1. `nptr->next = head;`

- First set the *next* node for `nptr` to the current **head**

2. `head = nptr;`

- Set the **head** node to be `nptr`

The operations for inserting at head need to be carried out in this order

- Think about if the operations were flipped
  - `head=nptr;`
  - `nptr->next = head;`
    - Except at this point you would have lost the original value of `head`

If you want to traverse backwards through a **linked list**, you can implement a **doubly linked list**