Today’s topics
- Linked lists, recursive data, intro to algorithm analysis & big-O

Reading
- linked lists Ch 9.5(sort of), handout #21
- algorithms, big O Ch 7

No cafe today after class :-(
- Due to undergrad council meeting

A recursive twist on printing
void PrintList(Entry *list)
{
    for (Entry *cur = list; cur != NULL; cur = cur->next)
        PrintEntry(cur);
}

Iteration replaced with recursion:
void PrintList(Entry *list)
{
    if (list != NULL) {
        PrintEntry(list);
        PrintList(list->next);
    }
}

What happens if we switch the order of these two lines?

Recursive data -> recursive ops
Natural to operate on linked list recursively
- List divides into first node and rest of list
- Base case: empty list
- Recursive case: handle first node, recur on rest

int Length(Entry *list)
{
    if (list == NULL) 
        return 0;
    else
        return 1 + Length(list->next);
}

void Deallocate(Entry *list)
{
    if (list != NULL) {
        Deallocate(list->next);
        delete list;
    }
}

void PrintEntry(Entry *entry)
{
    cout << entry->name << " " << entry->phone << endl;
}

void PrintList(Entry *list)
{
    for (Entry *cur = list; cur != NULL; cur = cur->next)
        PrintEntry(cur);
}

Idiomatic loop to iterate over list, compare to
for (int i = 0; i < n; i++)
Watch the pointers!

• (Decompose function to add node to front of list, mods shown in blue)

```c
void Prepend(Entry *ent, Entry *first)
{
    ent->next = first;
    first = ent;          // BUGGY!
}
```

```c
Entry *BuildAddressBook()
{
    Entry *listHead = NULL;
    while (true)  {
        Entry *newOne = GetNewEntry();
        if (newOne == NULL) break;
        Prepend(newOne, listHead);
    }
    return listHead;
}
```

Passing pointer by reference

• (Tiny modification in blue saves the day!)

```c
void Prepend(Entry *ent, Entry * &first)
{
    ent->next = first;
    first = ent;
}
```

Entry *BuildAddressBook()
{
    Entry *listHead = NULL;
    while (true)  {
        Entry *newOne = GetNewEntry();
        if (newOne == NULL) break;
        Prepend(newOne, listHead);
    }
    return listHead;
}

Array vs linked list

◊ Array/vector stores elements in contiguous memory
  • + Fast, direct access by index
  • - Insert/remove requires shuffling
  • - Cannot easily grow/shrink (must copy over contents)
◊ Linked list wires elements together using pointers
  • + Insert/remove only requires re-wiring pointers
  • + Each element individually allocated, easy to grow/shrink
  • - Must traverse links to access elements

Insert in sorted order

◊ Traverse list to find the position to insert
  ◊ What is true after the loop exits?

```c
void InsertSorted(Entry * &list, Entry * newOne)
{
    Entry *cur;
    for (cur=list; cur!= NULL; cur=cur->next)
    {
        if (newOne->name < cur->name) break;
    }
    cur->next = newOne;
    newOne->next = cur;
}
```

```
Kalev
123 Pine
555-0123
Lauri
204 Elm
555-8765
Matt
204 Elm
555-8765
Thomas
89 Oak
555-9425
Rein
19 Main
555-1234
```
Insert in sorted order

- Drag previous pointer (one behind cur)
  - prev/cur move down list in parallel, one node apart

```c
void InsertSorted(Entry * &list, Entry * newOne) {
    Entry *cur, *prev = NULL;
    for (cur=list; cur!= NULL; cur=cur->next){
        if (newOne->name < cur->name) break;
        prev = cur;
    }
    // what are possible values for prev?
    newOne->next = cur;     // splice outgoing ptr
    if (prev != NULL)
        prev->next = newOne; // splice incoming ptr
    else
        list = newOne;     // note special case!
}
```

Recursive insert

```c
void InsertSorted(Entry * & list, Entry * newOne)
{
    if (list == NULL|| newOne->name < list->name){
        newOne->next = list;
        list = newOne;
    } else {
        InsertSorted(list->next, newOne);
    }
}
```

- Wow!
  - Elegant, direct expression of algorithm
  - Dense use of pointers and recursion