CS 103 Unit 11

Linked Lists

Mark Redekopp
NULL Pointer

• Just like there was a null character in ASCII = '\0' whose value was 0

• There is a NULL pointer whose value is 0
  – NULL is "keyword" you can use in C/C++ that is defined to be 0
  – Used to represent a pointer to "nothing"
  – Nothing ever lives at address 0 of memory so we can use it to mean "pointer to nothing"

• `int* ptr = NULL; // ptr has 0 in it now`

• `if(ptr != NULL){ ... } // it's a good pointer`
Arrays Review

- Arrays are contiguous pieces of memory
- To find a single value, computer only needs
  - The start address
    - Remember the name of the array evaluates to the starting address (e.g. `data = 120`)
  - Which element we want
    - Provided as an index (e.g. `[20]`)
  - This is all thanks to the fact that items are contiguous in memory
  - If we know integer element `i` is at location 108 do we know where element `i+1` is?

```cpp
#include<iostream>
using namespace std;

int main()
{
    int data[25];
    data[20] = 7;
    return 0;
}
```
Limitations of Arrays

• We can dynamically allocate arrays once we know their size

• Example: Ask the user how many items they will need, then allocate an array for that size

• Problem: What if the user doesn’t know how many they will create or simply changes their mind
  – Example:
    • `cout << “How many numbers do you think you will input?” << endl; cin >> size;`
    • `int *ptr = new int[size];`
    • What if later the user wants to input an additional number??
    • Could allocate a new array of size+1 and copy items over but that becomes a time sink!

• Main point: Arrays, whether allocated statically or dynamically (using new or malloc), cannot be resized easily later on.
Analogy

• Natural analogy when we have a set of items that can change is to create a list
  – Write down what you know now
  – Can add more items later (usually to the end of the list)
  – Remove (cross off) others when done with them
• Can only do this with an array if you know max size of list ahead of time (which is sometimes fine)

1. Do CS 103 lab
2. Join ACM or IEEE
3. Play Video Games
4. Watch a movie
5. Exercise
6. Eat dinner
BFSQ Example

• The size of the BFSQ grew and shrunk based on the data pattern

• But we wasted a whole LARGE array planning for the worst case

• It'd be great to store only what we need where our storage can grow and shrink
Linked Lists

• A linked list stores values in separate chunks of memory (i.e. a dynamically allocated object)

• To know where the next one is, each one stores a pointer to the next

• We can allocate more or delete old ones as needed so we only use memory as needed

• All we do is track where the first object is (i.e. the head pointer)
Linked Lists

- What is the order of values in this linked list?
- How would you insert 6 at the front of the list?
- How would you remove the value 4?
Arrays vs. Linked List

- If we have the start address of an array can we get the i-th element quickly?
  - Yes: start-addr + i * sizeof(data)
- If we have the start (head) pointer to a linked list can we find the i-th element quickly?
  - No…Have to walk the linked list
- Linked lists trade the ability to resize (grow/shrink) for speed of access when attempting to get a specific element

```
#include<iostream>
using namespace std;

int main()
{
    int data[25];
    data[20] = 7;
    return 0;
}
```

```
data = 100
100 104 108 112 116 120
45 31 21 04 98 73 ...
```

```
head
0x148

0x148 0x1c0
3 next

0x148 0x1c0
9 next

0x168
2 (Null)
```
Linked List

- Use structures/classes and pointers to make ‘linked’ data structures
- List
  - Arbitrarily sized collection of values
  - Can add any number of new values via dynamic memory allocation
  - Usually supports following set of operations:
    - Append (“push_back”)
    - Prepend (“push_front”)
    - Remove back item (“pop_back”)
    - Remove front item (“pop_front”)
    - Find (look for particular value)

```cpp
#include <iostream>
using namespace std;

struct Item {
    int val;
    Item* next;
};

class List {
public:
    List();
    ~List();
    void push_back(int v); ...
private:
    Item* head;
};
```

**Rule of thumb**: Still use ‘structs’ for objects that are purely collections of data and don’t really have operations associated with them. Use ‘classes’ when data does have associated functions/methods.
Linked List

- Use structures/classes and pointers to make ‘linked’ data structures
- List
  - Arbitrarily sized collection of values
  - Can add any number of new values via dynamic memory allocation
  - Usually supports following set of operations:
    - Append (“push_back”)
    - Prepend (“push_front”)
    - Remove back item (“pop_back”)
    - Remove front item (“pop_front”)

```cpp
#include<iostream>
using namespace std;

List::List()
{
    head = NULL;
}

void List::push_back(int v){
    if(head == NULL){
        head = new Item;
        head->val = v; head->next = NULL;
    }
    else { ... }
}

int main()
{
    List mylist;
    mylist.push_back(3);
}
```

```
head
0x148

0x148

3 NULL

val next
```
Linked List

- Use structures/classes and pointers to make ‘linked’ data structures
- List
  - Arbitrarily sized collection of values
  - Can add any number of new values via dynamic memory allocation
  - Usually supports following set of operations:
    - Append (“push_back”)
    - Prepend (“push_front”)
    - Remove back item (“pop_back”)
    - Remove front item (“pop_front”)

```cpp
#include<iostream>
using namespace std;

List::List()
{
    head = NULL;
}

void List::push_back(int v){
    if(head == NULL){
        head = new Item;
        head->val = v; head->next = NULL;
    }
    else { ... }
}

int main()
{
    List mylist;
    mylist.push_back(3); mylist.push_back(9);
}
```

```
#include<iostream>
using namespace std;

List::List()
{
    head = NULL;
}

void List::push_back(int v){
    if(head == NULL){
        head = new Item;
        head->val = v; head->next = NULL;
    }
    else { ... }
}

int main()
{
    List mylist;
    mylist.push_back(3); mylist.push_back(9);
}
```
Linked List

- Use structures/classes and pointers to make ‘linked’ data structures
- List
  - Arbitrarily sized collection of values
  - Can add any number of new values via dynamic memory allocation
  - Usually supports following set of operations:
    - Append (“push_back”)
    - Prepend (“push_front”)
    - Remove back item (“pop_back”)
    - Remove front item (“pop_front”)

```cpp
#include<iostream>
using namespace std;

List::List()
{
    head = NULL;
}

void List::push_back(int v){
    if(head == NULL){
        head = new Item;
        head->val = v; head->next = NULL;
    }
    else { ... }
}

int main()
{
    List mylist;
    mylist.push_back(3); mylist.push_back(9);
    mylist.push_back(2);
}
```
Arrays Review

- Arrays are contiguous pieces of memory
- To find a single value, computer only needs
  - The start address
    - Remember the name of the array evaluates to the starting address (e.g. data = 120)
  - Which element we want
    - Provided as an index (e.g. [20])
  - This is all thanks to the fact that items are contiguous in memory
- Linked list items are not contiguous
  - Thus, linked lists have an explicit field to indicate where the next item is

```cpp
#include<iostream>
using namespace std;

int main()
{
    int data[25];
    data[20] = 7;
    return 0;
}
```
Common Linked Task/Mistake 1

- **What is the C++ code to take a step from one item to the next?**
- **Answer:**
  - `temp = temp->next`
- **Lesson:** To move a pointer to the next item use: 'ptr = ptr->next'
Common Linked Task/Mistake 2

- Why do we need a temp pointer? Why can't we just use head to take a step as in:
  - `head = head->next;`

- Because if we change head we have no record of where the first item is
  - Once we take a step we have "amnesia" and forget where we came from and can't retrace our steps

**Lesson:** Don't lose your head!
Common Linked Task/Mistake 3

- Mistake: Many students use the following code to get a pointer to the first item:
  - `Item* temp = head->next;`
- head (or first) pointer is NOT an actual ITEM struct
- head is just a pointer
  - It is special in that it is the only thing that is not actually holding any data...it just points at the first data-filled struct
  - head->next actually points to the 2\textsuperscript{nd} item, not the 1\textsuperscript{st} because head already points to the 1\textsuperscript{st} item
- **Lesson:** To get a pointer to the first item, just use 'head'
Common Linked Task/Mistake 4

- Common errors we see is that to create a temporary pointer students also dynamically allocate an item and then immediately point it at something else causing a memory leak
  - `Item* temp = new Item;`
  - `temp = head; or temp = head-&gt;next;`
- You may declare pointers w/o having to allocate anything
  - `Item* temp;`
  - `Item* temp = NULL;`
  - `Item* temp = head;`
- **Lesson:** Only use 'new' when you really want a new Item to come alive
Exercises

• In-class exercises:
  – monkey_traverse
  – monkey_addstart

Childs toy "Barrel of Monkeys" let's children build a chain of monkeys that can be linked arm in arm
Exercise

• Write an integer linked list class
• Download the skeleton:
  – Go to your examples directory
  – wget http://ee.usc.edu/~redekopp/cs103/listint.tar
  – tar xvf ListInt.tar
    • listint.h, listint.cpp, listint_test.cpp
• Examine the prototypes in listint.h (complete)
• Complete the functions in listint.cpp
• Compile and test your program the code in listint_test.cpp
Append

- Write a function to add new item to back of list
- Start from head and iterate to end of list
  - Copy head to a temp pointer
  - Use temp pointer to iterate through the list until we find the tail (element with next field = NULL)
  - Allocate new item and fill it in
  - Update old tail item to point at new tail item

I don’t know where the list ends so I have to traverse it
Remove First

- Write a function to remove first item
  - Copy address of first item to a temp pointer
  - Set head to point at new first item (only second item)
  - Deallocate old first item
Other Functions

• Write a function to print all items in list
  – Copy head to a temp pointer then use it to iterate over the items until the next pointer is NULL
  – Print each item as you iterate

• Find if an item in the list (return address of struct if present or NULL)
  – Copy head to a temp pointer then use it to iterate over the items until you find an item with the desired value or until next pointer is NULL

• Remove item with given value [i.e. find and remove]
  – If found, need to change the next link of the previous item to point at the item after the item found
Comparing Performance

Arrays
• Go to element at index I
  – O(1)
• Add something to the tail (assume you have a tail index)
  – O(1)
• Adding something to the front of the list after there are already n elements
  – O(n)

Linked Lists
• Go to element at index i
  – O(i)
• Add something to the tail (assume you have only head pointer and n elements in the list)
  – O(n)
• Adding something to the front of the list after there are already n elements
  – O(1)