CSCI 104
Linked Lists
Mark Redekopp
David Kempe
Array Problems

- Once allocated an array cannot grow or shrink
- If we don't know how many items will be added we could just allocate an array larger than we need but...
  - We might waste space
  - What if we end up needing more...would need to allocate a new array and copy items
- Arrays can't grow with the needs of the client

```plaintext
append(21) =>

Old, full array:

```
<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
<td>51</td>
<td>52</td>
<td>53</td>
<td>54</td>
<td>10</td>
</tr>
</tbody>
</table>
```

Allocate new array:

```
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
```

Copy over items:

```
<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
<td>51</td>
<td>52</td>
<td>53</td>
<td>54</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Add new item:

```
<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
<td>51</td>
<td>52</td>
<td>53</td>
<td>54</td>
<td>10</td>
<td>21</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
Motivation for Linked Lists

- Can we create a list implementation that can easily grow or shrink based on the number of items currently in the list?
- Observation: Arrays are allocated and deallocated in LARGE chunks
  - It would be great if we could allocate/deallocate at a finer granularity
- Linked lists take the approach of allocating in small chunks (usually enough memory to hold one item)
Linked List

- Use structures/classes and pointers to make ‘linked’ data structures
- A List is...
  - Arbitrarily sized collection of values
  - Can add any number of new values via dynamic memory allocation
  - Supports typical List ADT operations:
    - Insert
    - Get
    - Remove
    - Size
    - Empty
- Can define a List class

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

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

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

**Item blueprint:**

```
val
next
```

**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.
Don't Need Classes

• We don't have to use classes...
  – The class just acts as a wrapper around the head pointer and the operations
  – So while a class is probably the correct way to go in terms of organizing your code, for today we can show you a less modular, procedural approach

• Define functions for each operation and pass it the head pointer as an argument

    Item blueprint:
    ```
    int val
    Item* next
    ```

    #include<iostream>
    using namespace std;
    struct Item {
        int val;
        Item* next;
    };
    
    void append(Item*& head, int v);
    bool empty(Item* head);
    int size(Item* head);
    
    int main()
    {
        Item* head1 = NULL;
        Item* head2 = NULL;
        int sizel = size(head1);
        bool empty2 = empty(head2);
        ...
    }

    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 Implementation

- To maintain a linked list you need only keep one data value: **head**
  - Like a train engine, we can attach any number of 'cars' to the engine
  - The engine looks different than all the others
    - In our linked list it's just a single pointer to an Item
    - All the cars are Item structs
    - Each car has a hitch for a following car (i.e. next pointer)

```cpp
#include<iostream>
using namespace std;
struct Item {
    int val;
    Item* next;
};

void append(Item*& head, int v);

int main()
{
    Item* head1 = NULL;
    Item* head2 = NULL;
}
```

Engine = "head"  Each car = "Item"
A Common Misconception

• Important Note:
  – 'head' is NOT an Item, it is a pointer to the first item
  – Sometimes folks get confused and think head is an item and so to get the location of the first item they write 'head->next'
  – In fact, 'head->next' evaluates to the 2nd items address
Append

- Adding an item (train car) to the back can be split into 2 cases:
  - Attaching the car to the engine (i.e. the list is empty and we have to change the head pointer)
  - Attaching the car to another car (i.e. the list has other Items already) and so we update the next pointer of an Item

```cpp
#include<iostream>
using namespace std;
struct Item {
    int val;
    Item* next;
};

void append(Item*& head, int v) {
    if(head == NULL){
        head = new Item;
        head->val = v; head->next = NULL;
    }
    else {...}
}

int main() {
    Item* head1 = NULL;
    Item* head2 = NULL;
    append(head1, 3)
}
```
Linked List

- Adding an item (train car) to the back can be split into 2 cases:
  - Attaching the car to the engine (i.e. the list is empty and we have to change the head pointer)
  - Attaching the car to another car (i.e. the list has other Items already) and so we update the next pointer of an Item

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

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

void append(Item*& head, int v) {
    if(head == NULL){
        head = new Item;
        head->val = v; head->next = NULL;
    }
    else {...}
}

int main() {
    Item* head1 = NULL;
    Item* head2 = NULL;
    append(head1, 3)
}
```
Linked List

- Adding an item (train car) to the back can be split into 2 cases:
  - Attaching the car to the engine (i.e. the list is empty and we have to change the head pointer)
  - Attaching the car to another car (i.e. the list has other Items already) and so we update the next pointer of an Item

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

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

void append(Item*& head, int v) {
    if(head == NULL){
        head = new Item;
        head->val = v; head->next = NULL;
    } else {...}
}

int main() {
    Item* head1 = NULL;
    Item* head2 = NULL;
    append(head1, 3)
}
```
Append()

- Look at how the head parameter is passed...Can you explain it?
  - Head might need to change if it is the 1st item that we are adding
  - We've passed the head pointer BY VALUE so if we modify 'head' in append() we'll only be modifying the copy
  - We need to pass the pointer by reference
  - We choose Item* & but we could also pass an Item**

```cpp
void append(Item*& head, int v) {
    Item* newptr = new Item;
    newptr->val = v; newptr->next = NULL;
    if(head == NULL){
        head = newptr;
    } else {
        Item* temp = head;
        // iterate to the end ...
    }
}
```

```cpp
void append(Item** head, int v) {
    Item* newptr = new Item;
    newptr->val = v; newptr->next = NULL;
    if(*head == NULL){
        *head = newptr;
    } else {
        Item* temp = *head;
        // iterate to the end ...
    }
}
```
Arrays/Linked List Efficiency

- 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
  - This is "overhead" in terms of memory usage
  - Requires iteration to find an item or move to the end
Append()

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

```c
void append(Item*& head, int v) {
    Item* newptr = new Item;
    newptr->val = v; newptr->next = NULL;

    if(head == NULL) {
        head = newptr;
    } else {
        Item* temp = head;
        // iterate to the end ...
    }
}
```
Iterating Over a Linked List

• To iterate we probably need to create a copy of the head pointer (because if we modify 'head' we'll never remember where the list started)

• How do we take a step (advance one Item) given the temp pointer
  – temp = temp->next;

```cpp
void append(Item*& head, int v)
{
    Item* newptr = new Item;
    newptr->val = v; newptr->next = NULL;

    if(head == NULL){
        head = newptr;
    }
    else {
        Item* temp = head;
        while(temp->next){
            temp = temp->next;
        }
        temp->next = newptr;
    }
}
```
Using a For loop

void append(Item*& head, int v)
{
    Item* newptr = new Item;
    newptr->val = v; newptr->next = NULL;

    if(listPtr == NULL){
        head = newptr;
    }
    else {
        Item* temp = head;    // init
        while(temp->next){    // condition
            temp = temp->next;  // update
        }
        temp->next = newptr;
    }
}

void append(Item*& head, int v)
{
    Item* newptr = new Item;
    newptr->val = v; newptr->next = NULL;

    if(listPtr == NULL){
        head = newptr;
    }
    else {
        Item* temp;
        for(temp = head;        // init
            temp->next;         // condition
            temp = temp->next)    // update
        {
            temp->next = newptr;
        }
    }
}
void print(Item* head)
{
    Item* temp = head;  // init
    while(temp) {        // condition
        cout << temp->val << endl;
        temp = temp->next; // update
    }
}

void print(Item* head)
{
    Item* temp;
    for(temp = head;        // init
        temp;               // condition
        temp = temp->next){ // update
        cout << temp->val << endl;
    }
}
RECURSION & LINKED LISTS
Recursion and Linked Lists

- Notice that one Item's next pointer looks like a head pointer to the remainder of the linked list
  - If we have a function that processes a linked list by receiving the head pointer as a parameter we can recursively call that function by passing our 'next' pointer as the 'head'
Recursive Operations on Linked List

• Many linked list operations can be recursively defined
• Can we make a recursive iteration function to print items?
  – Recursive case: Print one item then the problem becomes to print the n-1 other items.
    • Notice that any 'next' pointer can be thought of as a 'head' pointer to the remaining sublist
  – Base case: Empty list (i.e. Null pointer)
• How could you print values in reverse order?

```c++
void print(Item* ptr)
{
    if(ptr == NULL) return;
    else {
        cout << ptr->val << endl;
        print(ptr->next);
    }
}

int main()
{
    Item* head;
    ...
    print(head);
}
```
Summing the Values

• Write a recursive routine to sum the values of a linked list
  – Head Recursion (recurse first, do work on the way back up)
  – Tail Recursion (do work on the way down, then recurse)
Head Recursion

- Recurse to the end of the chain (head == NULL) and then start summing on the way back up
  - What should the base case return?
  - What should recursive cases (normal nodes) return?

What would the prototype of this recursive function be?
Tail Recursion

• Produce sum as you walk down the list then just return the final answer back up the list

What would the prototype of this recursive function be?
Exercises

• Ilsum_head
• Ilsum_tail
Recursive Copy

• How could you make a copy of a linked list using recursion

```c
struct Item {
    int val;
    Item* next;
    Item(int v, Item* n){
        val = v; next = n;
    }
};

Item* copyLL(Item* head){
    if(head == NULL) return NULL;
    else {
        Item* copy = (Item*) malloc(sizeof(Item));
        copy->val = head->val;
        copy->next = copyLL(head->next);
        return copy;
    }
}

int main()
{ Item* oldhead, *newhead;
    ...
    newhead = copyLL(oldhead);
}
```
Recursive Copy

- How could you make a copy of a linked list using recursion

```c
struct Item {
    int val;
    Item* next;
    Item(int v, Item* n){
        val = v; next = n;
    }
};

Item* copyLL(Item* head)
{
    if(head == NULL) return NULL;
    else {
        return new Item(head->val, copyLL(head->next));
    }
}

int main()
{
    Item* oldhead, *newhead;
    ...
    newhead = copyLL(oldhead);
}
```
Recursive Copy

- How could you make a copy of a linked list using recursion

```c
struct Item {
    int val;
    Item* next;
    Item(int v, Item* n) {
        val = v; next = n;
    }
};

Item* copyLL(Item* head) {
    if (head == NULL) return NULL;
    else {
        return new Item(head->val, copyLL(head->next));
    }
}

int main() {
    Item* oldhead, *newhead;
    ...
    newhead = copyLL(oldhead);
}
```
INCREASING EFFICIENCY OF OPERATIONS + DOUBLY LINKED LISTS
Adding a Tail Pointer

• If in addition to maintaining a head pointer we can also maintain a tail pointer

• A tail pointer saves us from iterating to the end to add a new item

• Need to update the tail pointer when...
  - We add an item to the end (fast)
  - We remove an item from the end (slow)
Removal

- To remove the last item, we need to update the 2\textsuperscript{nd} to last item (set it's next pointer to NULL)
- We also need to update the tail pointer
- But this would require us to traverse the full list
- ONE SOLUTION: doubly-linked list
Doubly-Linked Lists

- Includes a previous pointer in each item so that we can traverse/iterate backwards or forward
- First item's previous field should be NULL
- Last item's next field should be NULL

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

struct DLItem {
    int val;
    DLItem* prev;
    DLItem* next;
};

int main()
{
    DLItem* head, *tail;
}
```
Doubly-Linked List Add Front

- Adding to the front requires you to update...
- ...Answer
  - Head
  - New front's next & previous
  - Old front's previous
Doubly-Linked List Add Front

- Adding to the front requires you to update...
  - Head
  - New front's next & previous
  - Old front's previous
Doubly-Linked List Add Middle

• Adding to the middle requires you to update...
  – Previous item's next field
  – Next item's previous field
  – New item's next field
  – New item's previous field
Doubly-Linked List Add Middle

• Adding to the middle requires you to update...
  – Previous item's next field
  – Next item's previous field
  – New item's next field
  – New item's previous field
Doubly-Linked List Remove Middle

- Removing from the middle requires you to update...
  - Previous item's next field
  - Next item's previous field
  - Delete the item object
Doubly-Linked List Remove Middle

• Removing from the middle requires you to update...
  – Previous item's next field
  – Next item's previous field
  – Delete the item object