Specification

This program will read in a list of cities from a file and store them in a Linked List. The program will have different methods to manipulate the list including traversing and destroying. Document the process with diagrams.

Analysis

- Inputs: File with names of cities in random order.
- Process: Read this cities from the file into a list using functions loadList, loadListInOrder, and buildDirectly. The list is printed then destroyed before calling the next load function
- Outputs: Display the list in each of the 3 different methods used to build it.

Design

- node: structure for the Nodes
- sll: structure for the single linked list including member functions
- insert: insert city into front of list
- insertInOrder: insert city in alphabetical order
- buildDirectly: insert city in the order of the cities.dat file
- displayList: prints out the list to the console
- destroyList: deletes each node in the list
- loadList: opens the file passed in and builds a list using insert
- loadListInOrder: opens the file passed in and build a list using inserInOrder
- buildDirectly: opens the file passed in and builds a list using buildDirectly
- main: the main function creates a SLL and passes it to the load functions

Implementation: lab.h File

This header file includes the three load functions required by the program along with various included libraries.

```
#include <string>
#include <iostream>
#include <fstream>
#include <vector>
#include "sll.h"

bool loadList(std::string, SLL*&);
bool loadListInOrder(std::string, SLL*&);
bool loadListDirectly(std::string, SLL*&);
```

Implementation: node.h File

This header file includes the structure for the node structure. The node has 2 data elements: the city, and the next pointer. There is also a constructor for quicker initialization of the data.

```
struct Node {
    std::string city;
    Node *next;

    Node(std::string city){
        this->city = city;
        this->next = nullptr;
    }
};
```

Implementation: sll.h File

This is the header file for the single linked list structure. The sll only has two data element which is the head pointer and the tail pointer. This file also includes the declarations for member functions used to manipulate the list.

```
#include "node.h"

struct SLL {

    Node *head;
    Node *tail;
    SLL(){
        this->head = nullptr;
        this->tail = nullptr;
    }
    SLL* insert(std::string city);
    SLL* insertInOrder(std::string city);
    SLL* build_directly(std::string city);
    SLL* display_list();
    SLL* destroy_list();
};
```

Implementation: insert.cpp File

This is the insert member function for SLL. It takes in a string as the city name. First it checks if it is an empty list and if it is, it will set that city as the head of the list. If there are nodes in the list, it will set the new cities next pointer to the current head node. Then it will set the head to the new actual head of the list. A list built with this method will be in reverse order.

```
#include "lab.h"

SLL* SLL::insert(std::string city) {

        Node *newNode = new Node(city);

        if (!this->head) {
            this->head = newNode;
        } else {
            Node *temp = this->head;
            this->head = newNode;
            newNode->next = temp;
        }

        return this;
    }
```

fig. 1

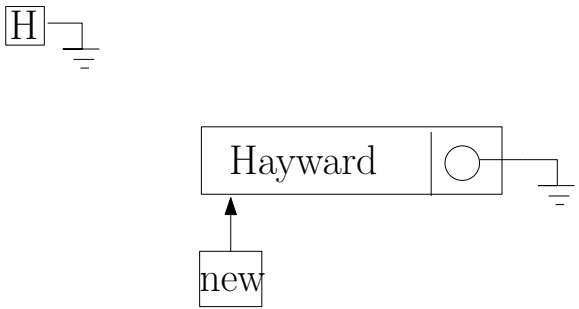


fig. 2

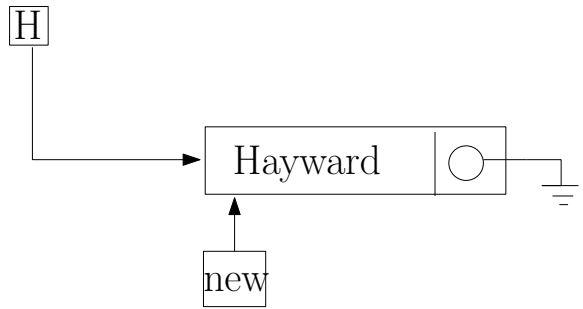


fig. 3

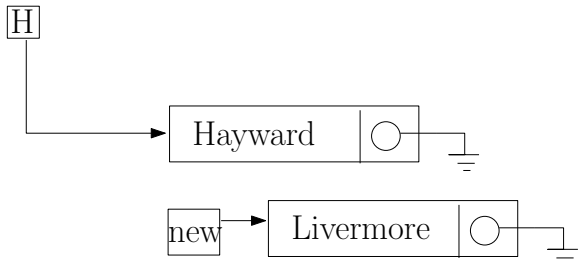


fig. 4

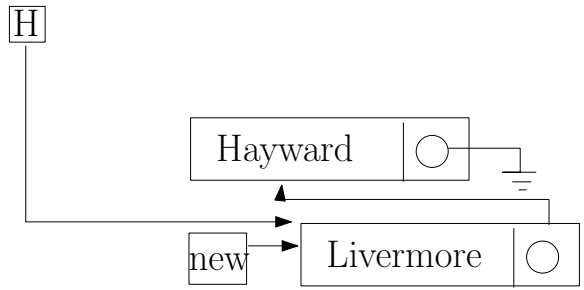


fig. 5

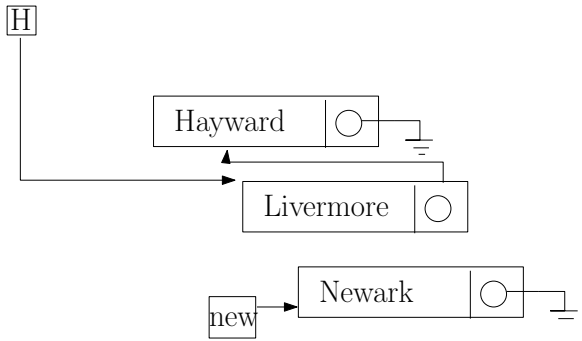

H
Hayward
Livermore
new
Newark

fig. 6

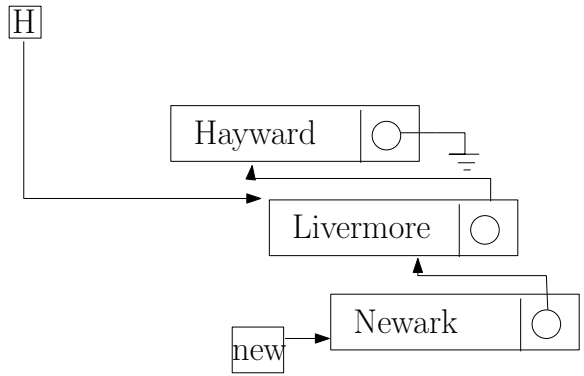
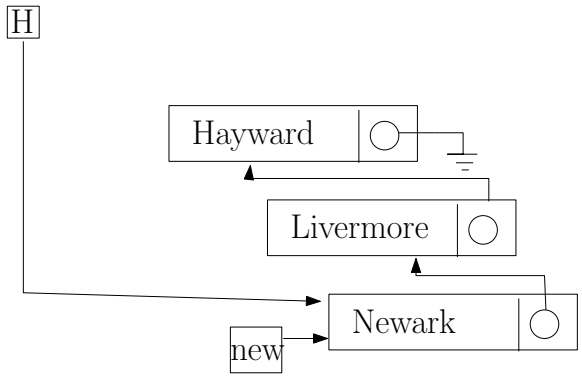

H
Hayward
Livermore
new
Newark

fig. 7



Implementation: insertInOrder.cpp File

This is the insert member function that will insert the new city in alphabatical order. This function uses a pointer to a Node pointer and traverse the list by following the pointers rather than the Nodes. The list will be in alphabetical order.

```
#include "lab.h"

SLL* SLL::insertInOrder(std::string city) {

    Node *newNode = new Node(city);
    Node **nodeptr = &head;

    while(*nodeptr && (*nodeptr)->city <= city){
        nodeptr = &((*nodeptr)->next);
    }

    newNode->next = *nodeptr;
    *nodeptr = newNode;

    return this;
}
```

fig. 1

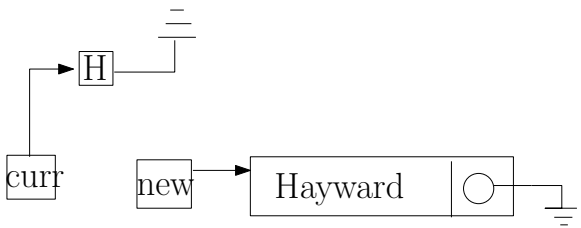


fig. 2

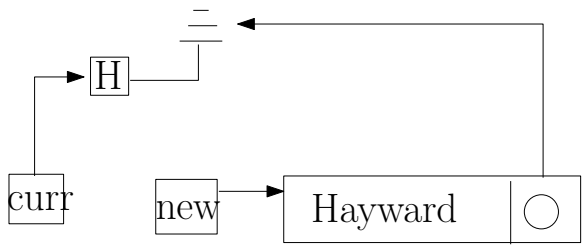


fig. 3

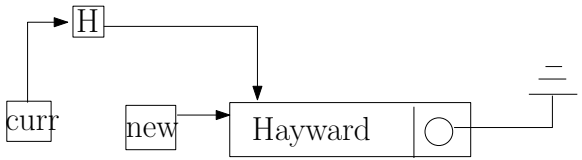


fig. 4

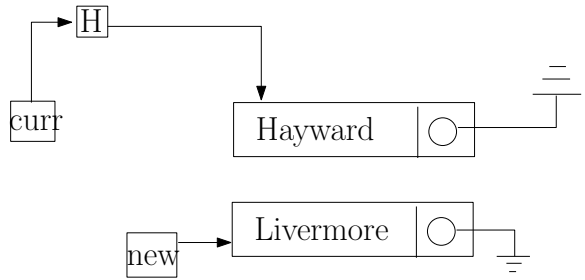


fig. 5

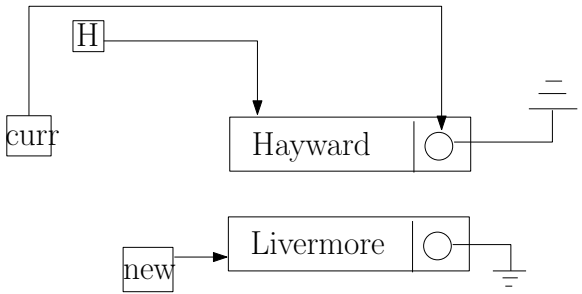


fig. 6

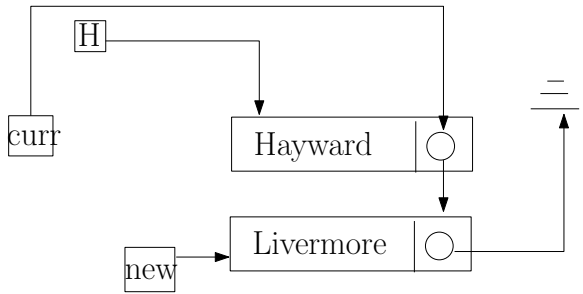


fig. 7

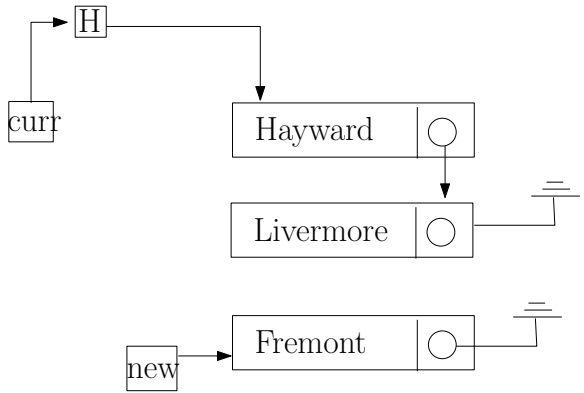

H
curr
Hayward
Livermore
Fremont
new

fig. 8

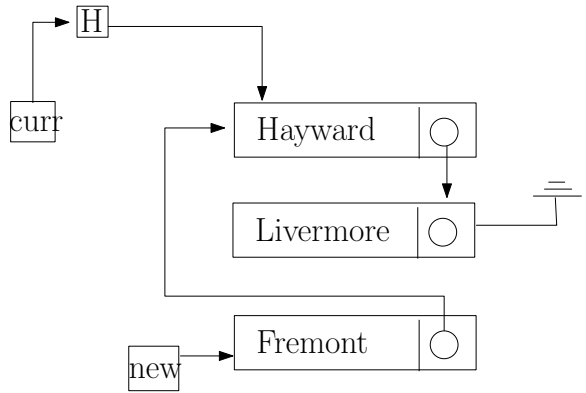
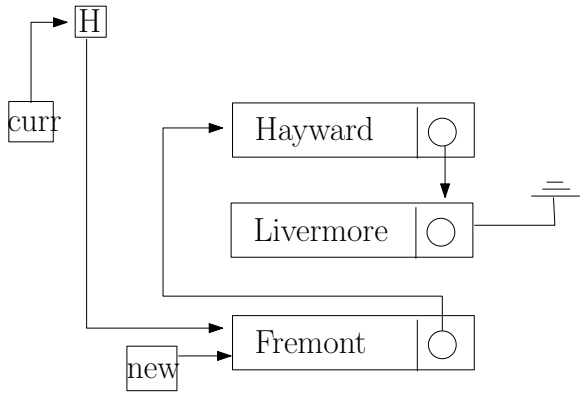

H
curr
Hayward
Livermore
Fremont
new

fig. 9



Implementation: buildDirectly.cpp File

This member function of the SLL struct will build the list using a tail pointer. This function is passed in the city and create a new node. It will set the current tail node to this new node. The tail will then be set to the new node added.

```
#include "lab.h"

SLL* SLL::build_directly(std::string city) {

    Node *newNode = new Node(city);

    if (!this->head) {
        this->head = newNode;
        this->tail = newNode;
    } else {
        this->tail->next = newNode;
        this->tail = newNode;
    }
        return this;
}
```

fig. 1

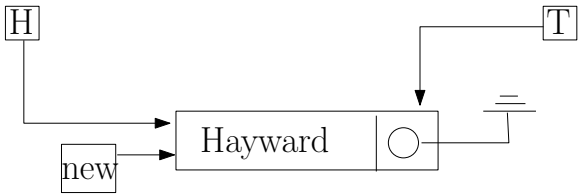


fig. 2

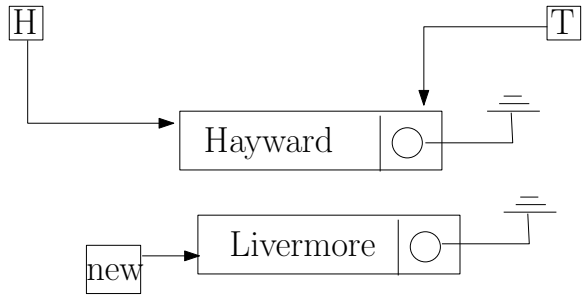


fig. 3

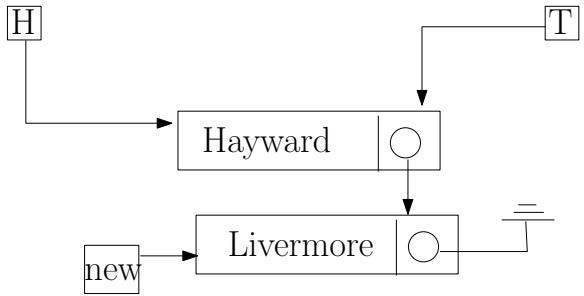


fig. 4

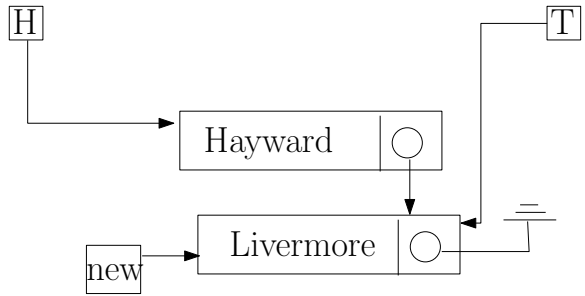


fig. 5

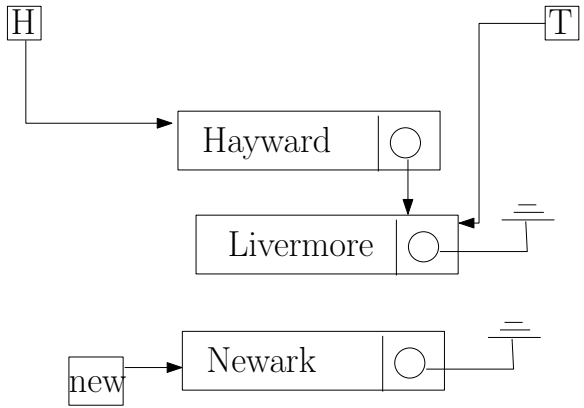

H
T
Hayward
Livermore
new
Newark

fig. 6

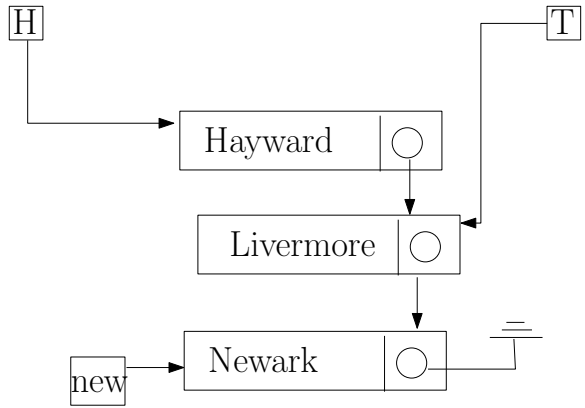
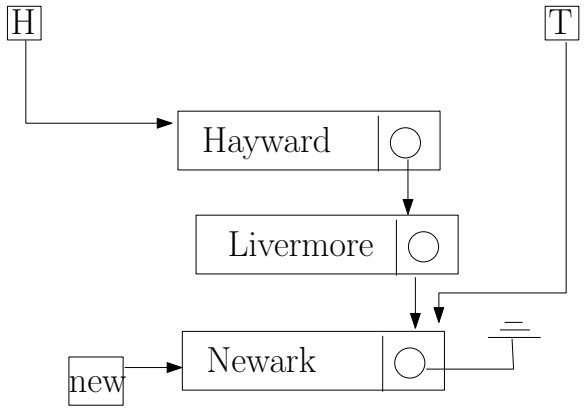

H
T
Hayward
Livermore
new
Newark

fig. 7



Implementation: displayList.cpp File

This method is used to print the cities to the console. It checks to see if the list is not empty. If the list is not empty, it will start at the head and print out its city element. Then it will follow the link of pointers to traverse the list.

```
#include "lab.h"

SLL* SLL::display_list() {

    if(this->head){
        Node *curr = this->head;

        while(curr) {
            std::cout << curr->city << std::endl;
            curr = curr->next;
        }
    }
        return this;
}
```

Implementation: destroyList.cpp File

This method is used to destroy the list by deleting every node in the list. Using a temp pointer to hold the remaining portion of the list, the function deletes the first node until there is none left. Finally it sets the head to nullptr.

```
#include "lab.h"

SLL* SLL::destroy_list(){

    Node* curr = this->head;

    while (curr) {
        Node* temp = curr->next;
        std::cout << "deleting " << curr->city << std::endl;
        delete curr;
        curr = temp;
    }

    this->head = nullptr;
    return this;

}
```

Implementation: loadList.cpp File

This function takes in a file name and a pointer to a list. The function opens the file cities.dat, returning false if it fails. If successful, the function will read in each line from the file and call the insert member function of the list. Finally it returns true if it is successful.

```
#include "lab.h"


bool loadList(std::string fname, SLL* &list){

    std::ifstream ifs(fname);
    if(!ifs)
        return false;

    std::string city;

    while(getline(ifs, city)){
        list->insert(city);

    }

    return true;

}
```

Implementation: loadListInOrder.cpp File

This function opens and reads from file the same as the loadList function. loadListInOrder calls the insertInOrder function rather than the insert function.

```
#include "lab.h"

bool loadListInOrder(std::string fname, SLL* &list){

    std::ifstream ifs(fname);
    if(!ifs)
        return false;

    std::string city;

    while(getline(ifs, city)){
        list->insertInOrder(city);

    }

    return true;
}
```

Implementation: loadListDirectly.cpp File

This function is same as the other two loadList functions with the only difference being that this one calls the buildDirectly function.

```
#include "lab.h"

bool loadListDirectly(std::string fname, SLL* &list){

    std::ifstream ifs(fname);
    if(!ifs)
        return false;

    std::string city;

    while(getline(ifs, city)){
        list->build_directly(city);
    }

    return true;


}
```

Implementation: lab.cpp File

This is the main function. It first creates a new SLL and then passes it to the loadList function. If the list is succesfully created, displayList will be called. The list is then destroyed by calling destroyList, and the list is rebuilt using the next method. When the list is built and destroyed with all 3 methods, the SLL is deleted and the program ends.

```
#include "lab.h"
int main(){
    SLL *city_list = new SLL();
    if(loadList("cities.dat", city_list)) {
        std::cout << "--Insert--" << std::endl;
        city_list->display_list();
        city_list->destroy_list();
    } else std::cout >> "Failed to load" >> std::endl;
    if(loadListInOrder("cities.dat",city_list)) {
        std::cout << "--In Order--" << std::endl;
        city_list->display_list();
        city_list->destroy_list();
    } else std::cout >> "Failed to load" >> std::endl;
    if(loadListDirectly("cities.dat",city_list)) {
        std::cout << "--Built Directly--" << std::endl;
        city_list->display_list();
        city_list->destroy_list();
    } else std::cout >> "Failed to load" >> std::endl;
    delete city_list;
}
```

Test

Test case that shows the three results. Insert should result in a list in the inverse as the cities.dat file. Insert in order should return a list that is in alphabetical order. Build Directly should give a list in the opposite order of the Insert method which will be the same order as the cities.dat file.

```
--Insert--
Milpitas
Oakland
Union City
Fremont
Newark
Livermore
Hayward
--In Order--
Fremont
Hayward
Livermore
Milpitas
Newark
Oakland
Union City
```

```
--Built Directly--
Hayward
Livermore
Newark
Fremont
Union City
Oakland
Milpitas
```