Data Structures

Linked List

The List Abstract Data Structure

- A list is made of the same kind of elements.
- Elements in the list:
  - $A_1, A_2, \ldots, A_N$
  - A_1: First element in the list
  - A_i: ith element in the list
- The list size is variable.
- The primary operations for the list:
  - printList: Print list elements to the screen.
  - makeEmpty: Delete all elements in the list.
  - find: Search for a given value in the list, find the related element.
  - insert: Insert data to the appropriate place in the list.
    - Insert to the beginning
    - Insert to the end
    - Insert to a particular position
  - remove: Find data and remove it from the list.

Example

List: 34, 12, 52, 16, 12

- find(52) Returns the number 3 which is the order of 52
- insert(X, 3) After this operation, the list looks like: 34, 12, X, 52, 16, 12
- remove(52) After this operation, the list looks like: 34, 12, X, 16, 12

- Operations and the functions that realize these operations could have a great variety.
- The inputs and outputs of every operation could be defined in different ways.

- insert(X,3) Add X as the 3. element.
- insert(X, 12) Add X after data 12.
- insert(X) Add X to the list in order.

List ADT with Array Implementation

- All operations we have defined can be realized using the array structure of C++.
- Problem: The array size has to be known in advance.

1. The array could be defined to have a constant size at the beginning of the program.

```
int A[100];
```

2. It could be defined to have a size given during execution.

```
int arraysize;
int *A;
cin << arraysize;
A = new int[arraysize];
```

- In both cases, the number of elements to be placed in the list has to known in advance.
- If defined to be unnecessarily big ⟹ waste of memory space
- If defined to be too small ⟹ insufficient space

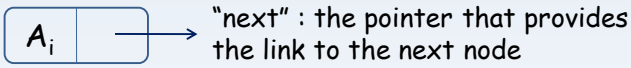
Importance of the Dynamic Structure

- printList and find are linear operations.
  - List elements are traversed in order from beginning to end.
- However, insert and remove are expensive operations to perform on the array.
  - To insert, all elements to the right of the inserted element should be shifted once to the right.
  - When the element is removed, the elements on the right should be shifted once to the left to get rid of the gap.

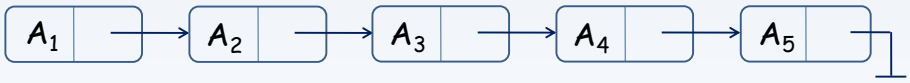
  Expensive = Too many element reads/writes (swaps) performed.
- Since the list structure requires these types of operations and can have variable size, using arrays to implement list structures is not preferred.

Linked List Implementation of the List ADT

- The linked list is made up of connected units.
- The units are called nodes.
- The type of each node is a structure made up of several fields; that is, it is a record.



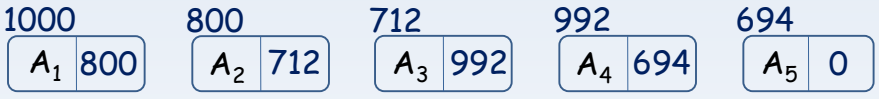
- Conceptual illustration of a linked list:



- The value of the next pointer of the last node is NULL.

Linked List Implementation of the List ADT

- The nodes of the linked list are not stored contiguously in memory!
  - As opposed to an array.



- To access a list, we have to know the address of the first node.
  - This address is 1000 above.
  - A special pointer is used as the list head pointer.

Phone Book

- We will see how the list operations can be performed on the phone book example introduced in the previous weeks.
- First, we have to make some changes in the program:
- At the beginning of phoneprog.cpp:

```
#include "list.h"
using namespace std;

typedef Phone_node Phone_Record;
typedef List Datastructure;
Datastructure book;
```

Header file that contains the structure where the list structure is defined

The nodes of the list structure are defined as Phone_node. The previous main program block uses Phone_Record. Phone_node will be used with the name Phone_Record.

The data structure will be List.

Changes to the Example

- A new option for clearing the list was added to the menu.

```
void print_menu(){
  cout << endl << endl;
  cout << "Phone Book Application" << endl;
  cout << "Select an operation" << endl;
  cout << "S: Record Search" << endl;
  cout << "A: Record Add" << endl;
  cout << "U: Record Update" << endl;
  cout << "D: Record Delete" << endl;
  cout << "C: Delete All" << endl;
  cout << "E: Exit" << endl;
  cout << endl;
  cout << "Enter an option {S, A, U, D, C, E} : ";
}
```


Functions

- No changes were made to the functions called by the main function:

```
void search_record();
void add_record();
void delete_record();
void update_record();
void clear_list();
```


New function

- In the phone book, the interfaces of the functions that enable operations also stayed the same, but of course their bodies will change.

```
void create();
void close();
void makeEmpty();
void insert(Phone_node *);
void remove(int ordernum);
int search(char *);
void update(int recordnum, Phone_node *);
```


New function

New Record Structure

```
#define NAME_LENGTH 30
#define PHONENUM_LENGTH 15

struct Phone_node{
  char name[NAME_LENGTH];
  char phonenum[PHONENUM_LENGTH];
  Phone_node *next;
};
```

- We have added the next pointer field that list nodes should have.

List Structure

```
#ifndef LIST_H
#define LIST_H
#include “node.h"

struct List{
  Phone_node *head;
  int nodecount;
  void create();
  void close();
  void printList();
  void makeEmpty();
  void insert(Phone_node *);
  void remove(int ordernum);
  int search(char *);
  void update(int recordnum, Phone_node *);
};
#endif
```

create()

```
void List::create (){
    head = NULL;
    nodecount = 0;
}
```

- Before starting to use the List structure, we should first initialize it.

```
typedef Phone_node Phone_Record;
typedef List Datastructure;

Datastructure book;
int main(){
    book.create();
...
```



```
int List::search(char *target){
  Phone_node *traverse;
  int counter = 0;
  int found = 0;
  traverse = head;
  bool all = false;
  if ( strcmp(target, "*") == 0 )
       all = true;

  while (traverse){
     counter++;
     if (all){
        cout << counter << "." << traverse->name << " " <<traverse->phonenum
                                                        <<endl;
        found++;
     }
     else if (strnicmp(traverse->name, target, strlen(target)) == 0){
        found++;
        cout << counter << "." << traverse->name << " " <<traverse->phonenum
                                                        <<endl;
     }
     traverse = traverse->next;
  }
  return found;
}
```

```
Phone Book Application
Choose an operation
S: Record Search
A: Record Add
U: Record Update
D: Record Delete
C: Delete All
E: Exit

Enter a choice (S, A, U, D, C, E) : s
Please enter the name of the person you want to search for (press '*' for full l
ist):
*
1.Gülşen 5324444444
2.Veli 2123333333
```


remove()

```
void List::remove(int
  ordernum){
  Phone_node *traverse, *tail;
  int counter = 1;
  traverse = head;
  if (ordernum <= 0){
       cout << "Invalid record
              order number.\n";
       return;
  }
  if (ordernum == 1){
       head = head->next;
       delete traverse;
       nodecount--;
       return;
  }
  while ((traverse != NULL) &&
         (counter < ordernum)){
     tail = traverse;
     traverse = traverse->next;
     counter++;
  }
  if (counter < ordernum){
  // given order num too large
     cout << "Could not find
       record to delete.\n";
  }
  else{ // record found
     tail->next = traverse->next;
     delete traverse;
     nodecount--;
  }
}
```

makeEmpty()

```
void List::makeEmpty(){
  Phone_node *p;
  while (head){
      p = head;
      head = head->next;
      delete p;
  }
  nodecount = 0;
}
```


insert()

```
void List::insert(Phone_node
        *toadd){
  Phone_node *traverse, *tail;
  Phone_node *newnode;
  traverse = head;
  newnode = new Phone_node;
  *newnode = *toadd;
  newnode->next = NULL;
  if (head == NULL){
     //first node being added
        head = newnode;
        nodecount++;
        return;
  }
  if (strcmp(newnode->name, head->name) < 0){
        //Insert to head of list
        newnode->next = head;
        head = newnode;
        nodecount++;
        return;
  }
  while (traverse &&
  (strcmp(newnode->name, traverse->name) > 0)){
        tail = traverse;
        traverse = traverse->next;
  }
  if (traverse){ // Insert into a position
        newnode->next = traverse;
        tail->next = newnode;
  }
  else // Insert to end
        tail->next = newnode;
  nodecount++;
}
```

update()

```
void List::update(int recordnum, Phone_node *newnode){
  Phone_node *traverse;
  int counter = 1;
  traverse = head;
  while ( traverse && (counter < recordnum) ){
       counter++;
       traverse = traverse->next;
  }
  if (traverse){
       newnode->next = traverse->next;
       *traverse = *newnode;
  }
  else
       cout << "Invalid number for record to be
               updated.\n";
}
```


End of Program

- When the program is being ended, all the space allocated for dynamic data structures has to be returned to the system.
- The records in the phone book are held in a linked list.
- When the program is ending, all nodes must be deleted.

```
int main(){
  book.create();
  bool end = false;
  char choice;
  while (!end) {
    print_menu();
    cin >> choice;
    end = perform_operation(choice);
    }
  book.close();
  return EXIT_SUCCESS;
}
```

```
void List::close(){
      makeEmpty();
}
```

Making Data Permanent

- After the program has been closed, the data has to be stored in the hard disk so that it is not lost.
- That is why the records are saved to a file when closing the program in our lecture example.

```
struct List{
        ...
        char *filename;
        FILE *phonebook;
        void read_fromfile();
        void write_tofile();
};
```



```
void List::create(){
  head = NULL;
  nodecount = 0;
  read_fromfile();
}

void List::close(){
  write_tofile();
  makeEmpty();
}
```

```
void List::read_fromfile(){
  struct File_Record{
       char name[NAME_LENGTH];
       char phonenum[PHONENUM_LENGTH];
  };
  File_Record record;
  Phone_node *newnode;
  filename = "phonebook.txt";
  if ( !(phonebook = fopen( filename, "r+" ) ) )
       if ( !(phonebook = fopen( filename, "w+") ) ){
              cerr << "File could not be opened" << endl;
              exit(1);
       }
  fseek(phonebook, 0, SEEK_SET);
  while ( !feof(phonebook) ){
       newnode = new Phone_node;
       fread( &record, sizeof (File_Record), 1, phonebook);
       if (feof(phonebook)) break;
       strcpy(newnode->name, record.name);
       strcpy(newnode->phonenum, record.phonenum);
       newnode->next = NULL;
       insert(newnode);
  }
  fclose(phonebook);
}
```



```
void List::write_tofile(){
  struct File_Record{
       char name[NAME_LENGTH];
       char phonenum[PHONENUM_LENGTH];
  };
  File_Record record;
  Phone_node *p;
  if ( !(phonebook = fopen( filename, "w+" ) ) ){
              cerr << "File could not be opened" << endl;
              exit(1);
  }
  p = head;
  while (p){
       strcpy(record.name, p->name);
       strcpy(record.phonenum, p->phonenum);
       fwrite(&record,sizeof(File_Record), 1, phonebook);
       p = p->next;
  }
  fclose(phonebook);
}
```