Data Structures

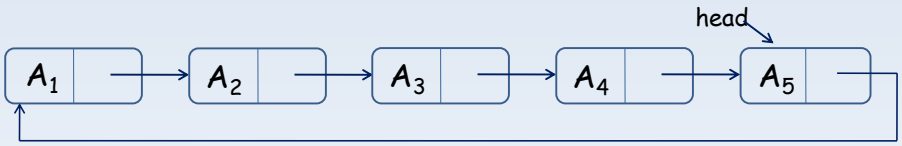
Implementations and Types of Lists

Circular List

Circular List

- The circular list is a widely used list implementation.
- A link is established from the last node to the first node.
- The "next" field of the last node is not NULL, but instead points to the first node.
- Advantage:
  - Starting from any node in the list, we can traverse to the end of the list and get back to the beginning.
- We do not need to make any changes in the definitions (the node and list structures).
- How we determine that we have reached the end of the list will change.
- In general, the bodies of functions such as insert() and remove() that perform list operations will change.

Circular List

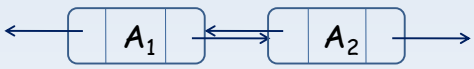


- In a circular list, the list head pointer must point to the end of the list.
  - Thus, both the beginning of the list and the end of the list can be reached in one step.
  - head points to the last node in the list.
  - head->next points to the first node in the list.
  - This makes it easier to insert to and remove from the beginning and to insert to the end.

Doubly Linked List

Doubly Linked List

- List nodes contain both forward and backward links.



- The list can be traversed in both directions by following the pointers.
- We have to make changes to the list operations.
- We will make the following changes to the design of the list:
  - The node structure will change.
  - The list structure will contain both head and tail pointers.
  - The bodies of list operations will change.

Node Structure

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


List Structure

```
struct List{
  Phone_node *head, *tail;
  int nodecount;
  char *filename;
  FILE *phonebook;
  void create();
  void close();
...
};
```

create()

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


insert()

```
void List::insert(Phone_node *toadd){
   Phone_node *traverse, *newnode;
   traverse = head;
   newnode = new Phone_node;
   strcpy(newnode->name, toadd->name);
   strcpy(newnode->phonenum, toadd->phonenum);
   newnode->previous = NULL;
   newnode->next = NULL;
   if (head == NULL) { // first node is being added
         head = newnode;
         tail = newnode;
         nodecount++;
         return;
   }
   if ( strcmp(newnode->name, head->name) < 0 ) { // add to head of list
         newnode->next = head;
         head = newnode;
         (newnode->next)->previous = newnode;
         nodecount++;
         return;
   }
```

insert() (continued)

```
  while ( traverse && (strcmp(newnode->name, traverse->name) > 0) ) {
        // newnode's name comes after traverse's
        traverse = traverse->next;
  }
  if (traverse) { // insert in between
        newnode->next = traverse;
        newnode->previous = traverse->previous;
        (traverse->previous)->next = newnode;
        traverse->previous = newnode;
  }
  else{ // insert to end
        tail->next = newnode;
        newnode->previous = tail;
        tail = newnode;
  }
  nodecount++;
}
```


remove()

```
void List::remove(int ordernum){
  Phone_node *traverse;
  int counter = 1;
  traverse = head;

  if (ordernum <= 0) {
       cout << "Invalid record order number.\n";
       return;
  }

  if (ordernum == 1){
       head = head->next;
       head->previous = NULL;
       delete traverse;
       nodecount--;
       return;
  }
```

remove() (continued)

```
  while ( (traverse != NULL) && (counter < ordernum) ) {
       traverse = traverse->next;
       counter++;
  }
  if (counter < ordernum) { // given record num too large
       cout << "Could not find record to delete.\n";
  }
  else { //record found
       (traverse->previous)->next = traverse->next;
       if (traverse->next)
             traverse->next->previous = traverse->previous;
       else
             tail = traverse->previous;
       delete traverse;
       nodecount--;
  }
}
```


search() and update()

- No changes have to be made to these functions.
- Since the search() function starts from the head of the list and searches the list going forward, `previous` fields are not used.
- Same is true for update().

Multilist Example

Designing the Data to Suit the Program

- Many variations could be created on the linked list basic structure.
- The programmer must determine the most suitable structure for the program at hand.
- When the suitable structure is selected, the best data storage and access environment has been prepared for writing the program.
- Before starting to write a program, the data must designed carefully.
- Data is the fundamental building block of a program. When data is designed correctly, writing, debugging, and testing the program becomes easier.

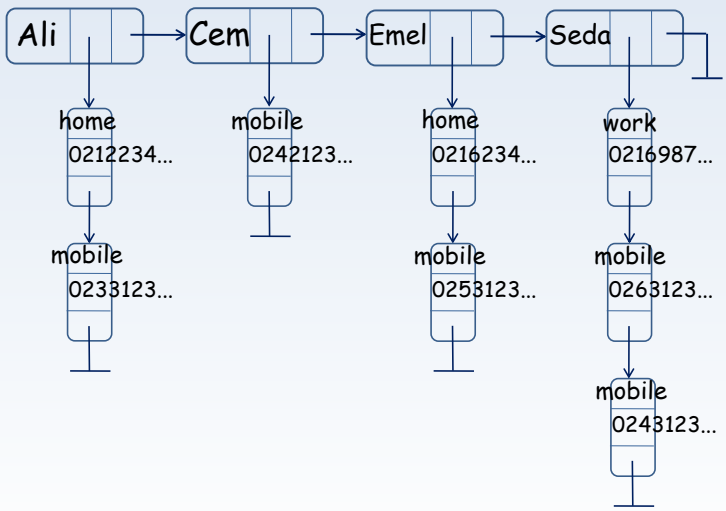
Types of Linked Lists

- The linked structure could be used for creating multidimensional lists: Multilist
- In multilists, more than one list is constructed using different node types.
- The data and pointer types to be contained in each node type are designed based on the structure.

A New Design for the Phone Book

- Our phone book example actually does not take into account the case of each person having more than one phone number.
- Solution:
  - The main list will be made up of nodes that hold the name of a person and a pointer to the list that contains the phone numbers of that person.
  - Phone numbers belonging to each person will be kept in a separate list along with phone types (work, home, mobile).
- We are designing a two-dimensional structure.

Multilist Structure for Phone Book



Node Structures

```
#define NAME_LENGTH 30
#define PHONENUM_LENGTH 15
#define TNAME_LENGTH 4

struct number{
    char type[TNAME_LENGTH];
    char num[PHONENUM_LENGTH];
    number *next;
};

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

List Structure

```
struct List{
   Phone_node *head;
   int personcount;
   char *filename;
   FILE *phonebook;
   void create();
   Phone_node *create_node(char *, char *, char *);
   void close();
   void makeEmpty();
   void insert(char *, char *, char *);
   void remove(int ordernum);
   int search(char *);
   void update(int, char *);
   void read_fromfile();
   void write_tofile();
};
```

Changes in declaration and body
Changes in body

makeEmpty()

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

create_node()

```
Phone_node * List::create_node(char *name, char *phone,
                                           char *type){
    Phone_node *newperson;
    number *newnum;
    newperson = new Phone_node;
    strcpy(newperson->name, name);
    newnum = new number;
    newperson->phonenum = newnum;
    strcpy(newnum->num, phone);
    strcpy(newnum->type, type);
    newnum->next = NULL;
    newperson->next = NULL;
    return newperson;
}
```


insert()

```
void List::insert(char *newname, char *newphone, char *newtype){
  Phone_node *traverse, *behind, *newperson;
  number *newnum;
  traverse = head;
  if (head == NULL) { // first node being added
       head = create_node(newname, newphone, newtype);
       personcount++;
       return;
  }
  if ( strcmp(newname, head->name) < 0 ) { // add to head of list
       newperson = create_node(newname, newphone, newtype);
       newperson->next = head;
       head = newperson;
       personcount++;
       return;
  }
```

insert() (continued)

```
        while ( traverse && (strcmp(newname, traverse->name) > 0) ) {
            // newname comes after traverse's name
            behind = traverse;
            traverse = traverse->next;
        }
        if (traverse && strcmp(newname, traverse->name) == 0){
            // this name was added before; just add phone number
            newnum = new number;
            newnum->next = traverse->phonenum;
            traverse->phonenum = newnum;
            strcpy(newnum->num, newphone);
            strcpy(newnum->type, newtype);
        }
        else {
            newperson = create_node(newname, newphone, newtype);
            if (traverse) { // inserting new name in between
                    newperson->next = traverse;
                    behind->next = newperson;
            }
            else // insert to the end
                    behind->next = newperson;
            personcount++;
      }
   }
```


remove()

```
void List::remove(int ordernum){
  Phone_node *traverse, *behind;
  number *pn;
  int counter = 1;
  traverse = head;
  if (ordernum <= 0) {
        cout << "Invalid record order number.\n";
        return;
  }
  if (ordernum == 1) {
        head = head->next;
        pn = traverse->phonenum;
        while (pn) {
                traverse->phonenum = pn->next;
                delete pn;
                pn = traverse->phonenum;
        }
        delete traverse;
        personcount--;
        return;
  }
```

remove() (continued)

```
   while ( (traverse != NULL) && (counter < ordernum) ){
        behind = traverse;
        traverse = traverse->next;
        counter++;
   }
   if (counter < ordernum){ // given record num too large
        cout << "Could not find record to delete.\n";
   }
   else{ // record found
        behind->next = traverse->next;
        pn = traverse->phonenum;
        while (pn) {
           traverse->phonenum = pn->next;
           delete pn;
           pn = traverse->phonenum;
        }
        delete traverse;
        personcount--;
   }
}
```


search()

```
int List::search(char *target){
   Phone_node *traverse;
   number *pn;
   int counter = 0;
   int found = 0;
   traverse = head;
   bool all = false;

   if (target[0] == '*')
       all = true;
```

search() (continued)

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


update()

- Updates only the name.

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

read_fromfile()

```
void List::read_fromfile(){
   struct File_Record{
         char name[NAME_LENGTH];
         char phonenum[PHONENUM_LENGTH];
         char type[TNAME_LENGTH];
   };
   File_Record record;
   filename = "phonebook.txt";
   if ( !(phonebook = fopen( filename, "r+" ) ) ) {
         if ( !(phonebook = fopen( filename, "w+" ) ) ) {
                  cerr << "File could not be opened." << endl;
                  cerr << "Will work in memory only." << endl;
                  return;
         }
   }
   fseek(phonebook, 0, SEEK_SET);
   while ( !feof(phonebook) ) {
         fread(&record, sizeof(File_Record), 1, phonebook);
         if ( feof(phonebook) ) break;
         insert(record.name, record.phonenum, record.type);
   }
   fclose(phonebook);
}
```


write_tofile()

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