Phone Directory Application using Linked Lists

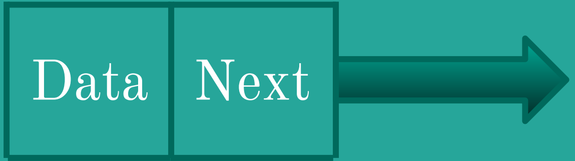
A Mini Project on Data Structure (CS 201) for End Semester examination of Odd Semester, 2020-21

Submitted by:
1914089, 1914092, 1914097, 1914099, 1914101, 1914102, 1914105, 1914127, and 1914173
3rd Semester, UG (BTech) Scholars
Department of Electronics and Communication Engineering
National Institute of Technology Silchar

A General Introduction to Linked Lists and its application

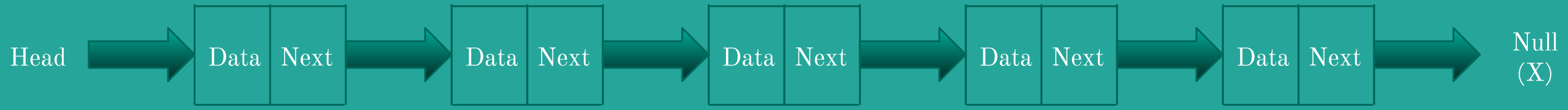
- A linked list is a linear data structure with each of its elements called a node.
- These nodes, which don’t occupy contiguous memory locations.
- They are used store the data as well as a pointer to point to the next node
- The next pointer creates a link between two adjacent nodes.
- Linked lists are created with the help of the help of structures.
  Consider the following piece of code:

```
typedef struct node{
        int data;
        struct node *next;
}node;
```



A node

The above piece of code defines a node for the linked list where the *next pointer is used to point to the next node thereby creating a link between the two.



A linked list formed of five nodes, with *head pointer pointing to the first node, and the last node's *next pointer pointing to NULL

- Analogous to the above sample of a linked list, the use of linked list can be expanded to the case of phone directory.
- Each contact in the directory can be represented by a node.
- Each node can contain as many data fields as required and a pointer *next to point the next contact, i.e., the next node.
- The use of linked list allows essential data like name, with phone number, and also additional information like address, relation with the owner, etc.
- The mini project has been an attempt to realise the same.

The Components of the Directory

The definition of a node to store each contact:

```
typedef struct node{ // linked list to store contacts, also referred to as ‘contact’ list
        long long number[2] ;
        char name[50] ;
        char address[100] ;
        char relation[20] ;
        struct node* next ;
}node ;
```

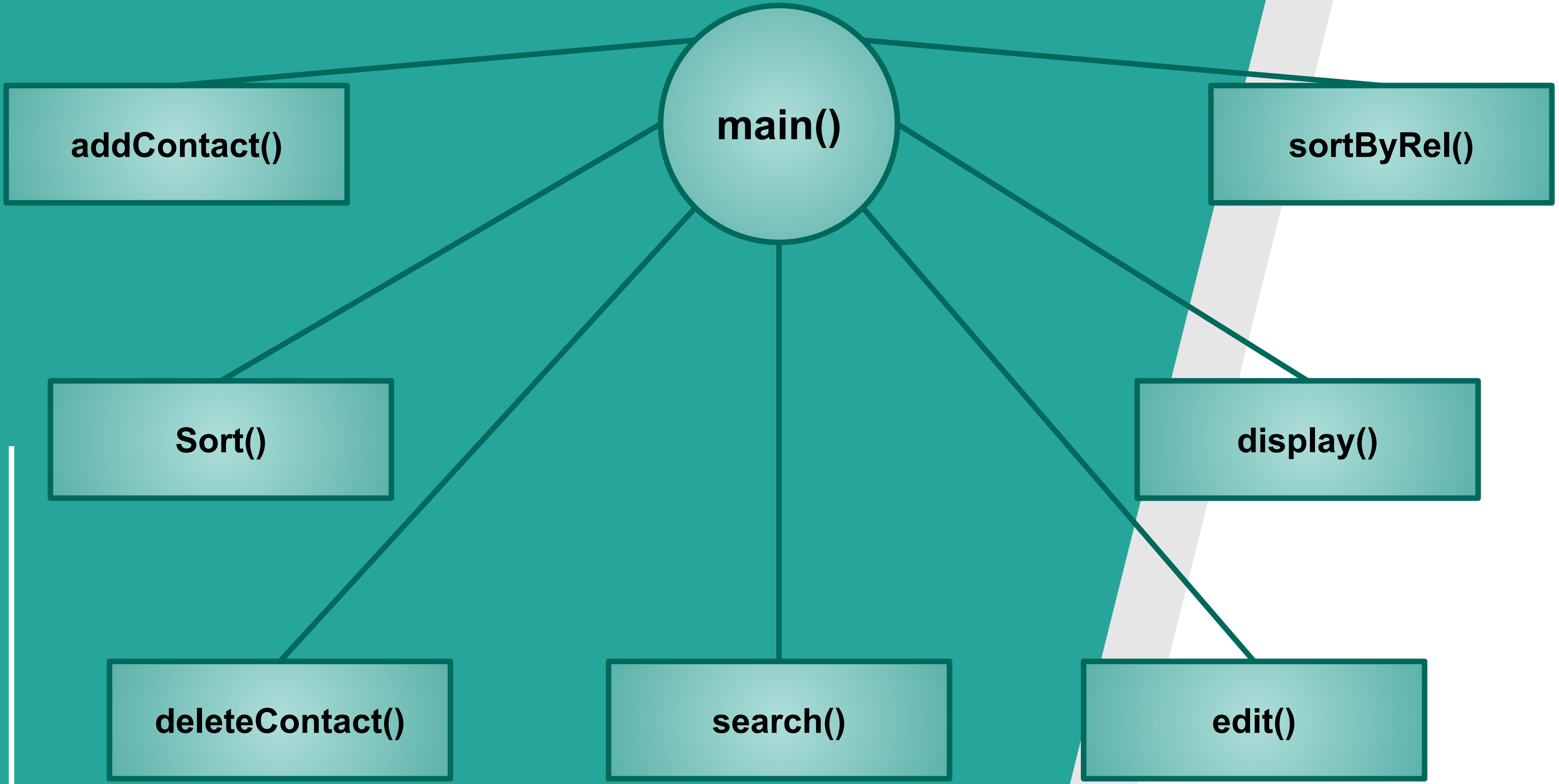
- The above snippet is used to define a node for a contact, having the data fields of a name, two phone numbers, an address, and a relation.
- The pointer *next is used to point the next contact stored in the subsequent node.

The definition of a node to store a group:

```
typedef struct grp { // linked list to sort by relation, also referred to as ‘relation’ list
        char grp_name[50] ;
        node *front;
        struct grp* next ;
}grp ;
```

- The above snippet is used to define a node for a group, having the data fields of a name, a node pointer to point the first contact stored within the group.
- The pointer *next is used to point the next group stored in the subsequent node.

The Functions of the Directory


main()
addContact()
sortByRel()
Sort()
display()
deleteContact()
search()
edit()

1. **void addContact() :**

- Used to create a phone contact which is to be saved by the user;
- The Contact contains Name, Address, Phone Number and Relationship with user.;
- Each Name Must be unique;
- If the Name entered by the User already exists in the directory, then it shows an error, and user must re-enter the Name for the New Contact;
- Then the data is inserted to the end of the ‘contact’ list
- It is also inserted to the end of the list formed by ‘relation’ node with the same ‘relation’

2. **void Sort(&head) :**

- This function sorts the name of the people whose names have been saved in the directory.
- The associate function that work for this functions are:

  void split(h, &a, &b)

  This function uses the technique of merge sort algorithm to divide the array into smallest part then merge the sorted part, for sorting the names.
- The **void Sort(&head)** function is called again and again to until the names in the directory are sorted.
- The names which have been sorted are merged together to form the complete name, using the function:

  node* merge(a, b)

3. **void deleteContact() :**

- Used to delete the contact details of a person whose name is entered
- If the name is not present in the phone directory, prints an error message
- Otherwise, it removes the contact of that person
- Both the ‘contact’ list node and the node from the list of the ‘relation’ node is removed

4. **void search() :**

- Used to search the details of the person whose name goes as input.
- If the name of that person does not exist in the phone directory, the programme prints "Name does not exist in Phone directory! Please enter a valid name or press 0 to exit!”
- If ‘0’ is entered, the operation terminates.
- Otherwise the details of that person is printed

5. **void edit() :**

- Used to edit the details of the person whose name was given in input
- If the name does not exist in the phone directory, prints an error message
- Otherwise it asks for the following options -
    A. Edit name ?
    B. Edit number ?
    C. Edit address ?
    D. Edit relation ?
    E. Exit ?
- If the input is different from (A,B,C,D,E) prints an error message
- Otherwise, the new value provided overwrites the old one.

6. **void display() :**

- In case of empty directory, prints "Empty Directory!"
- Accesses each and every contact by traversing with a node pointer from the very first node to the last one
- Prints the details of the contact being accessed
- Doesn’t return anything

7. **void sortByRel() :**

- This function is used to sort the contacts based on the information stored in their ‘relation’ field.
- Accesses each group (relation) starting from the first one
- In case of absence of any group, prints "Empty Directory!"
- Prints the name of the group being accessed
- Accesses each contact stored in the group (relation)
- Prints the details of the contact being accessed
- No additional function is used within this function
- Doesn’t return anything

8. **void printDetails(node* t) :**

- This function prints all the details of the node pointed by the pointer ‘t’
- It Prints the
  - Name
  - Address
  - Phone number(s)
  - Relation

9. **void copy_string(char a[], char b[]) :**

- A function that copies the content of string a to string b

10. **node* getNewNode(long long n[], char c1[], char c2[], char c3[]) :**

- This function returns a ‘contact’ node with the phone numbers initialized from n[], name from c1[], address from c2[], relation from c3[].

11. **grp* createGrp(char c[]) :**

- This function returns ‘relation’ node with the ‘relation’ as c[] and all the pointers as ‘NULL’

13. **void insert(long long n[], char c1[], char c2[], char c3[]) :**

- Used to insert a node in the 'contact' list at the end

14. **int find_grp(char c[], grp **ptr) :**

- Returns 1, if a 'relation' node equal to c[] is found, otherwise 0

15. **void grp_insert(long long n[], char c1[], char c2[], char c3[]) :**

- Used to insert a 'contact' node in the list of a 'relation' node

16. **int alreadyExistName(char c[], node** ptr) :**

- Returns 1 if the c is present in the 'name' field of the 'contact' list, otherwise 0

17. **int same_name(char a[], char b[]) :**

- Returns 1 if ‘a’ and ‘b’ have the same length and each character of ‘a’ and ‘b’ at same index are same

18. **int numberAlreadyExist(long long n) :**

- Returns 1 if the n is present in the ‘number’ field of the ‘contact’ list, otherwise 0

19. **int cmp(char a[], char b[]) :**

- Is used to compare two strings and sort in alphabetical order
- Returns 0 if a is lexicographically smaller or equal to b
- Otherwise returns 1
- The comparison is not case sensitive

20. **void split(node* source, node** frontRef, node** backRef) :**

- Is used to carry out the merge sort algorithm for sorting the list
- Uses two pointers *fast and *slow to access the nodes of a linked list
- *fast traverses until and unless *fast reaches the end of the list and in each step traverses two nodes at once
- *slow traverses one node at a time
- Splits the linked list into two halves by setting the *next of *slow as NULL

21. **node* merge(node* a, node* b) :**

- Is used to merge the nodes which have been sorted after being split
- In case of either of the two nodes being NULL, the function returns the other node
- Compares the two nodes based on their names using the int cmp(char a[], char b[]) function
- Sets a node result with the node being lexicographically smaller amongst the two
- Recursively, merge is called to compare and merge the next node
- int cmp(char a[], char b[]) function is used within this function
- Returns the node result

GLOBAL VARIABLES :

- grp* grp_head - points to the start of ‘relation’ list
- node* head - points to the start of ‘contact’ list

THANK YOU!