NAMIBIA UNIVERSITY OF SCIENCE AND TECHNOLOGY

Data structures and Algorithms

Group Project

Name	Student number
Penehafo Nakale	223138339
Ndilipawa Alweendo	223031488
Isron E Ndaningina	223031992
Jacinto C Tchayevala	224084003
Anesu A Mwape	224044095
Esegel S Narib	223086770

Table of Content

INSERTION OF CONTACT PSEUDOCODE & FLOWCHART

front = null

rear = null

enqueue(contact){

newNode -> data =contactac

newNode -> next = null

IF(front == null AND rear == null)THEN

front = newNode

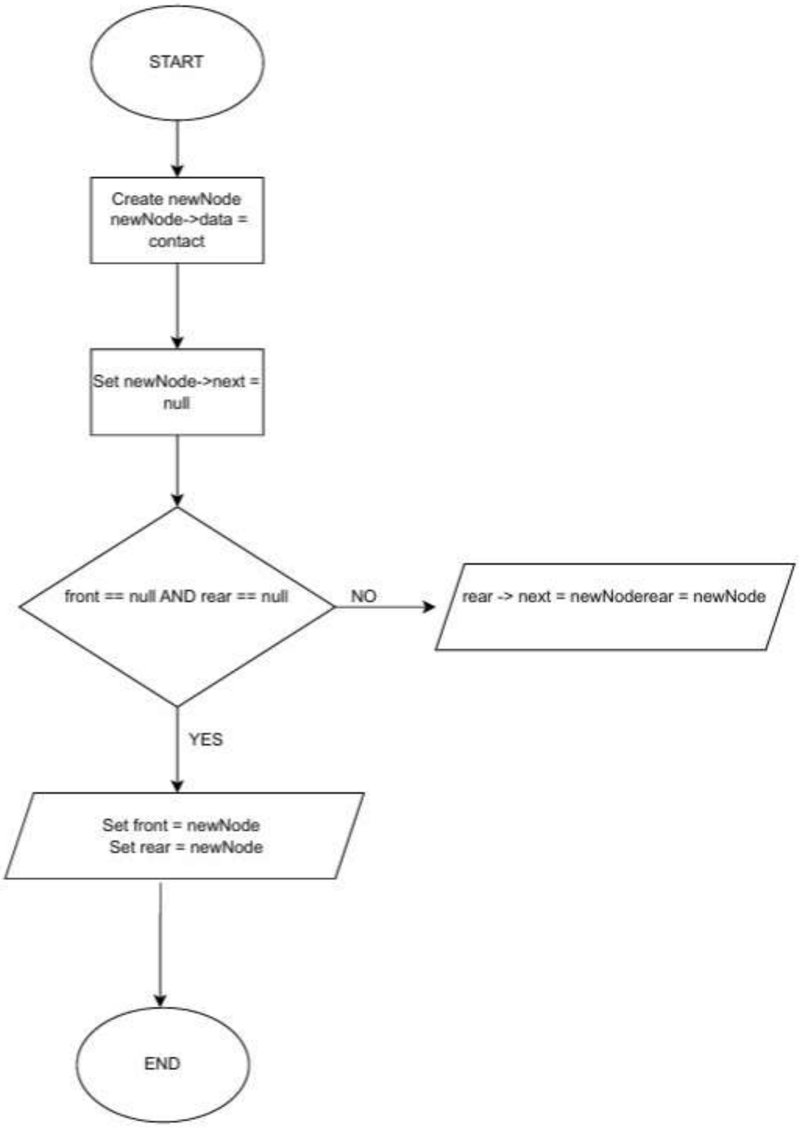
rear = newNode

ELSE

rear -> next = newNoderear = newNode

ENDIF

}


START
Create newNode
newNode->data =
contact
Set newNode->next =
null
front == null AND rear == null
NO
rear -> next = newNoderear = newNode
YES
Set front = newNode
Set rear = newNode
END

SEARCH PSEUDOCODE & FLOWCHART

Start

index = hashFn(letter)

If (array[index] == null)

array[index] = (letter, number)

Else

while (array[index] != null and array[index][0] != letter)

index = (index + 1) % arraySize

if (array[index] != null and array[index][0] == letter)

return array[index][1]

else

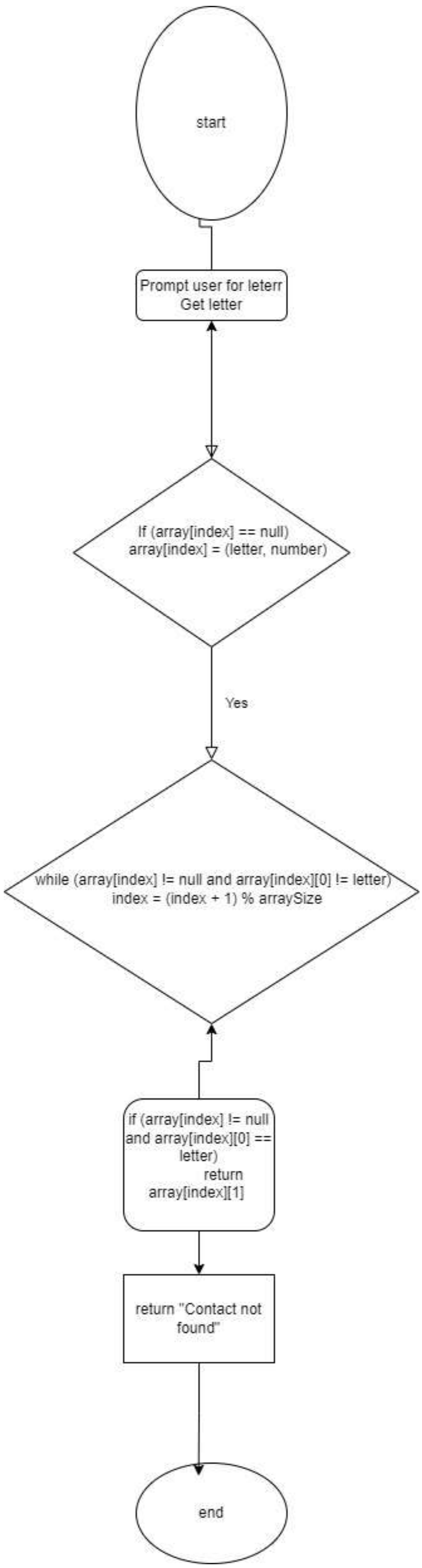
return "Contact not found"

End

End if

End if

End


start
Prompt user for leterr
Get letter
If (array[index] == null)
array[index] = (letter, number)
Yes
while (array[index] != null and array[index][0] != letter)
index = (index + 1) % arraySize
if (array[index] != null and array[index][0] == letter)
return array[index][1]
return "Contact not found"
end

DISPLAY ALL CONTACTS PSEUDOCODE & FLOWCHART

delete(){

temp = front

IF(front == null AND rear == null)THEN

DISPLAY “phonebook is empty”

ELSE

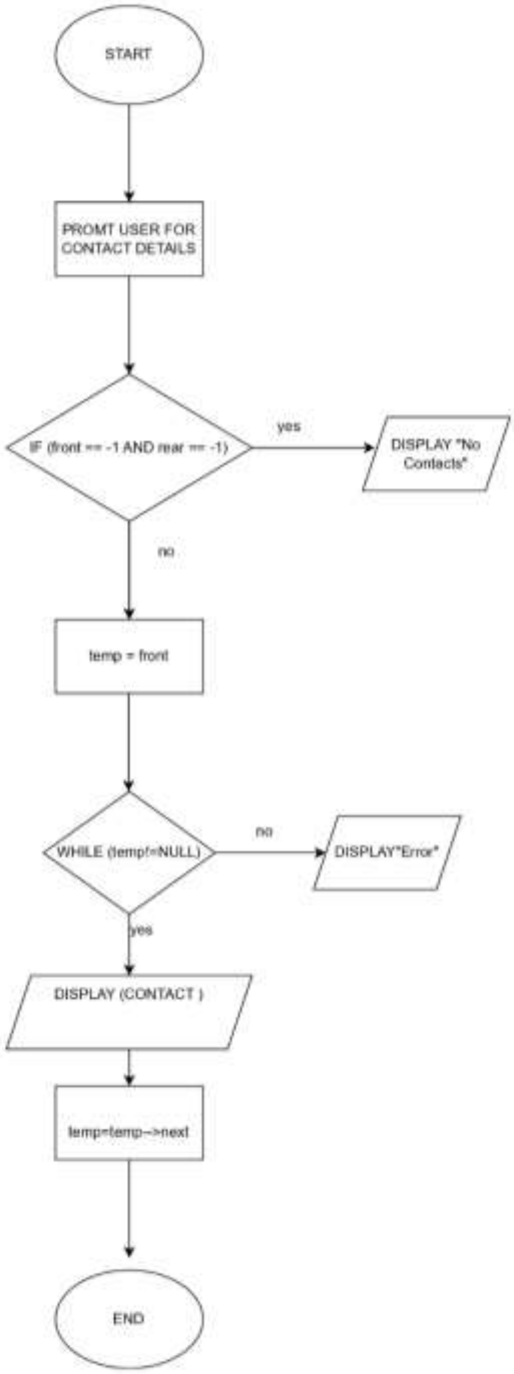
DISPLAY “The deleted contact is: ” front -> data

front = front -> next

temp = free

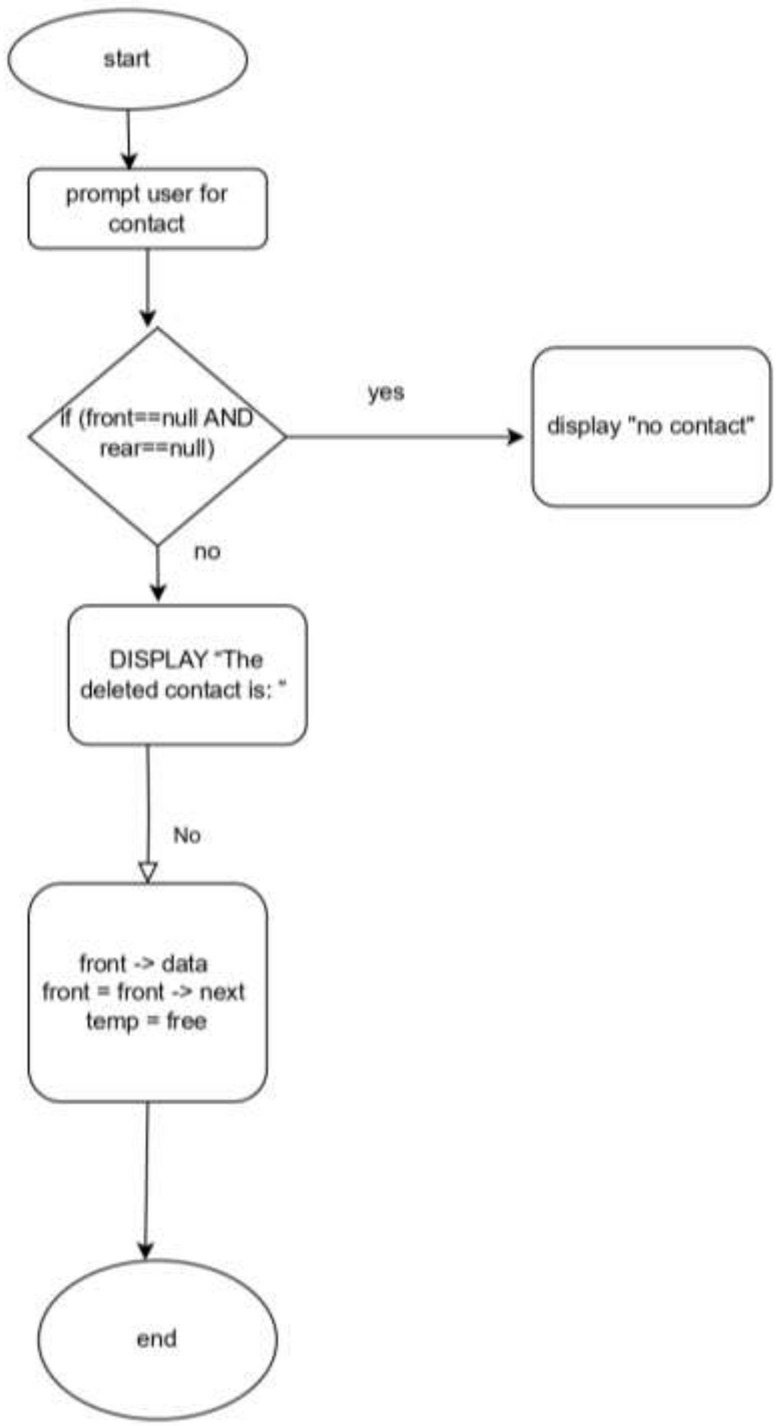
ENDIF

}


START
PROMT USER FOR CONTACT DETAILS
IF (front == -1 AND rear == -1)
yes
DISPLAY "No Contacts"
no
temp = front
WHILE (temp!=NULL)
no
DISPLAY"Error"
yes
DISPLAY (CONTACT)
temp=temp-->next
END

DELETE CONTACT PSEUDOCODE & FLOWCHART

```
delete(){
temp = front
IF(front == null AND rear == null)THEN
DISPLAY “phonebook is empty”
ELSE
DISPLAY “The deleted contact is: ” front -> data
front = front -> next
temp = free
ENDIF
}
```


start
prompt user for contact
if (front==null AND rear==null)
yes
display "no contact"
no
DISPLAY "The deleted contact is: "
No
front -> data
front = front -> next
temp = free
end

UPDATE CONTACT PSEUDOCODE & FLOWCHART

updateContact(String name, String newPhoneNumber)

Create contactFound as Boolean = False

Create tempQueue as new LinkedList<Contact>( )

While contactQueue is not empty

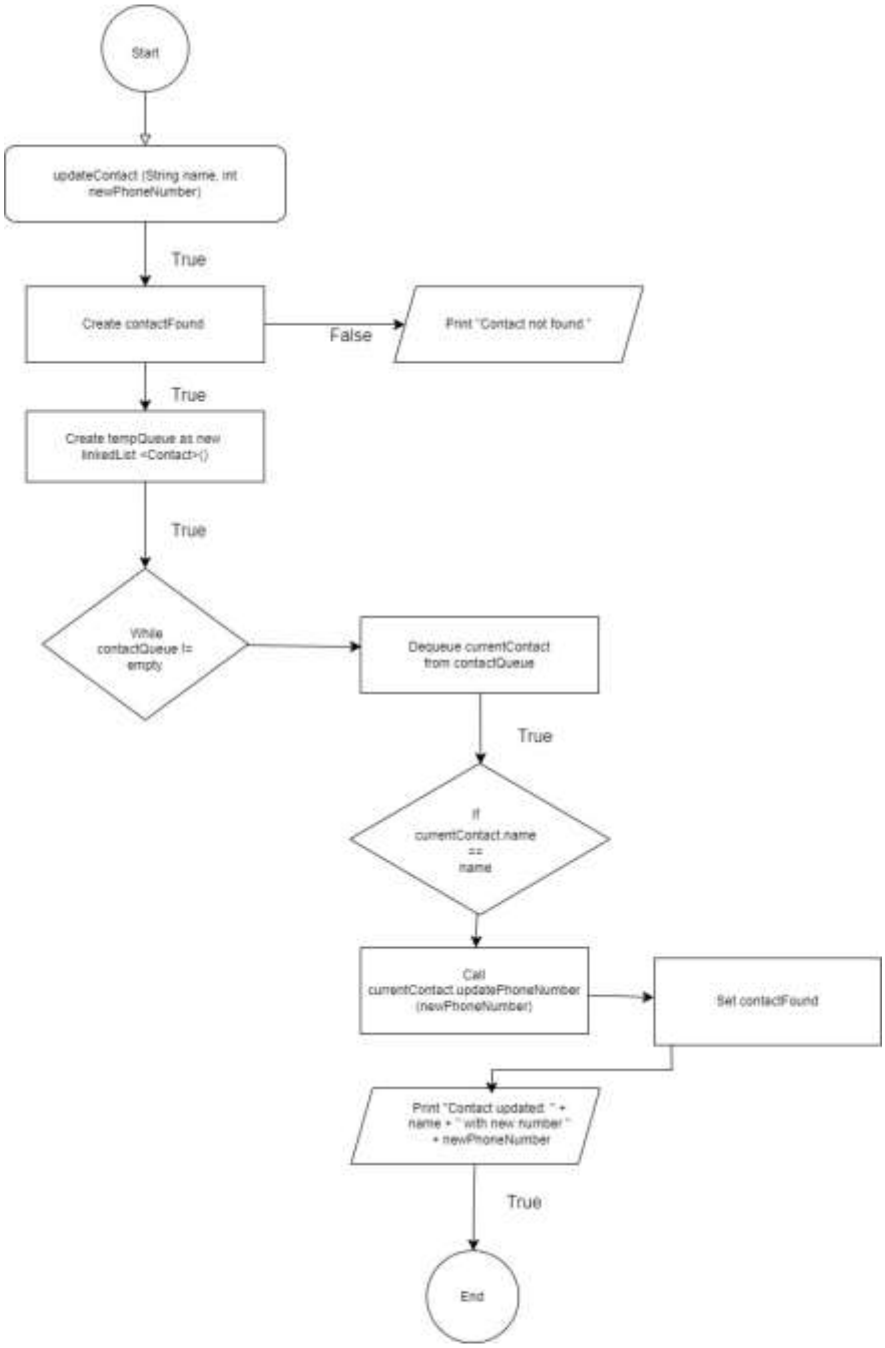
Dequeue currentContact from contactQueue

If currentContact.name == name

Call currentContact.updatePhoneNumber(newPhoneNumber)

Set contactFound = True

Print "Contact updated: " + name + " with new number " + newPhoneNumber


Start
updateContact (String name, int newPhoneNumber)
True
Create contactFound
False
Print "Contact not found."
True
Create tempQueue as new linkedList <Contact>()
True
While contactQueue != empty
Dequeue currentContact from contactQueue
True
If currentContact.name == name
Call currentContact.updatePhoneNumber (newPhoneNumber)
Set contactFound
Print "Contact updated: " + name + " with new number " + newPhoneNumber
True
End

SORT CONTACTS PSEUDOCODE & FLOWCHART

sortContacts()

Create tempStack as new LinkedList<Contact>()

While contactStack is not empty

Pop(currentContact from contactStack)

While(tempStack is not empty AND tempStack.peek().name > currentContact.name)

Pop (tempContact from tempStack)

Push (tempContact to contactStack)

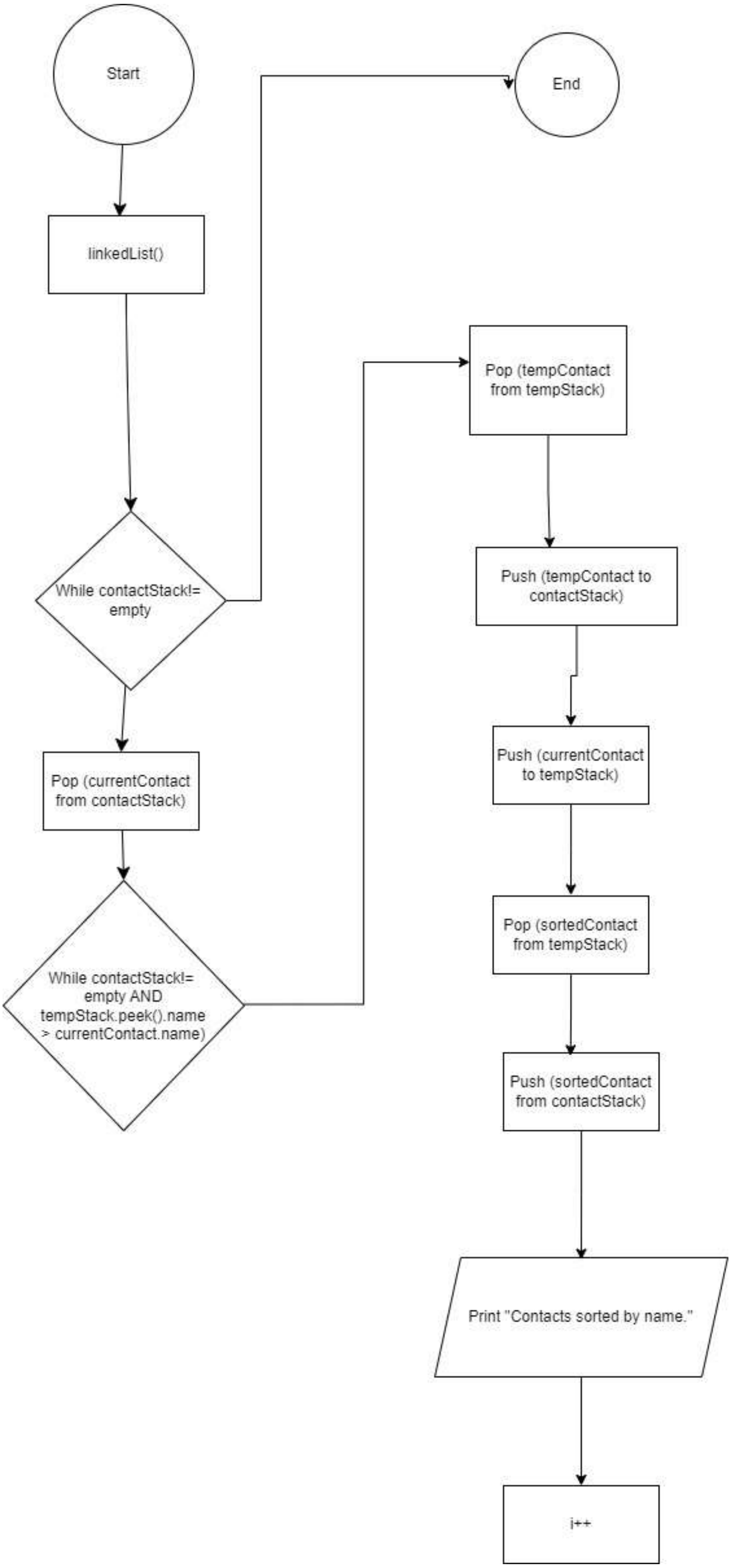
Push(currentContact to tempStack)

While tempStack is not empty

Pop (sortedContact from tempStack)

Push(sortedContact to contactStack)

Print "Contacts sorted by name."


Start
End
linkedList()
Pop (tempContact from tempStack)
While contactStack!= empty
Push (tempContact to contactStack)
Pop (currentContact from contactStack)
Push (currentContact to tempStack)
While contactStack!= empty AND tempStack.peek().name > currentContact.name)
Pop (sortedContact from tempStack)
Push (sortedContact from contactStack)
Print "Contacts sorted by name."
i++

SEARCH ALGORITHM ANALYSIS

<u>Search Algorithm Efficiency</u>

In our phonebook application, we employed a hash table for efficient contact retrieval, significantly enhancing search performance.

<u>How Hash Tables Work</u>

A hash table uses a hash function to map keys (in this case, contact names) to specific indices in an array. This allows for direct access to the contact data based on the hash value derived from the contact's name. When a user searches for a contact, the application computes the hash value of the input name and quickly locates the corresponding index in the hash table.

<u>Efficiency Analysis</u>

1. *<u>Time Complexity</u>*:

- *Best Case*: O(1) - When the hash function distributes keys evenly, searches can be completed in constant time.
- *Average Case*: O(1) - Under typical conditions, the average time for a successful search remains constant.
- *Worst Case:* O(n) - If many contacts hash to the same index (a collision), the application may need to check each entry at that index, leading to linear time complexity. However, good hash functions minimize collisions, making this scenario rare.

2. *<u>Space Complexity</u>*:

The hash table consumes more memory compared to a simple list due to its underlying array structure. However, this trade-off is justified by the significant speed advantage during search operations.

3. *<u>Practical Considerations</u>*:

- The application efficiently manages up to several thousand contacts, making it well-suited for personal use or small business applications.

- The user experience is streamlined, as searches return results almost instantaneously, allowing for a smooth interaction.

In summary, the use of a hash table in our phonebook application optimizes the search process, making it not only faster but also user-friendly. This implementation demonstrates the balance between memory usage and performance, ensuring that users can efficiently manage their contacts without delay.