Data Structures and Algorithms

W2- Lecture 1,2

Array and List Data Structure

Engr. Bushra Tahir
Department of Electrical Engineering
Iqra National University

Arrays

- An array is a data structure that is a collection of variables of one type that are accessed through a common name.
- A specific element is accessed by an index.
- Array can hold multiple values of a single type.
- Elements are referenced by the array name and an ordinal index
- Indexing begins at zero.
- The name of the array holds the address of the first array element.

Linear Arrays

- The simplest form of array is a one-dimensional array that may be defined as a finite ordered set of homogeneous elements, which is stored in contiguous memory locations.
- An array may contain all integers or all characters or any other data type, but may not contain a mix of data types.
- *Example:* If we choose the name A for the array, then the elements of A are denoted by subscript notation:

a1, a2, a3, ……., an

Linear Arrays

- Parenthesis notation:

  A(1), A(2), A(3),……., A(N) or,
- bracket notation:

  A[1], a[2], A[3],…….., A[N]
- number K in A[K] is called a subscript and A[K] is called a subscripted variable.
- Linear arrays are called one-dimensional arrays because each element in such an array is referenced by one subscript.

Array Declaration

The general form for declaring a single dimensional array is:

```
data_type array_name [expression]
```

- Data type represents data type of the array. That is, integer, char, float etc.
- Array name is the name of array
- Expression which indicates the number of elements in the array

Example

```
int a[100];
```

It declares an array of 100 integers.

Array Storage

- The amount of storage required to hold an array is directly related to its type and size.
- For a single dimension array, the total size in bytes required for the array is computed as

Memory required (in bytes) = size of (data type) X length of array

Array Initialization

- Initializing an array while declaring it.

```
int a[4] = {34,60,93,2};
int b[] = {2,3,4,5};
float c[] = {-4,6,81,” 60};
```

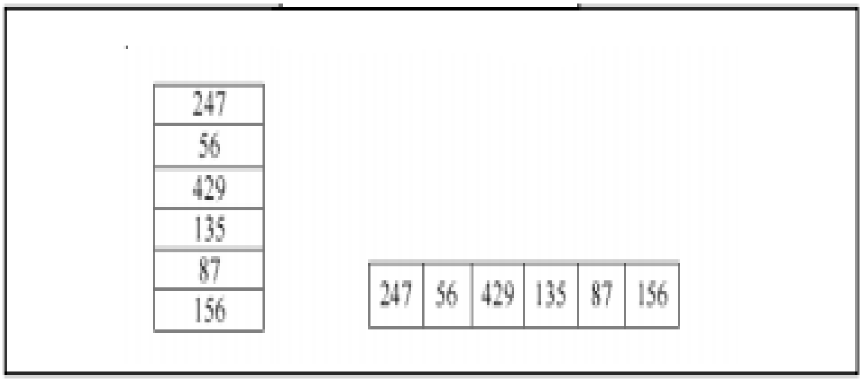
- If the array is initialized at the time of declaration, then the dimension of the array is optional.
- Till the array elements are not given any specific values, they contain garbage values.

Example

Let us consider.

Array DATA be a 6-element linear array of integers such that

DATA[1]=247, DATA[2]=56, DATA[3]=429, DATA[4]=135, DATA[5]=87, DATA[6]=156.



Example

Let us consider

- x = 3; //not allowed
- x = a + b; // not allowed
- x = &n; // not allowed

Example

```
int*  y = new int[20];
```

- It means we are requesting computer to find twenty memory locations
- Now *y* has become an array and we can say *y[0] =1 or y[5] = 15*.
- New returns the memory address of first of the twenty locations and we store that address into *y*.

Multidimensional Array

- Arrays with more than one dimension are called multi-dimensional arrays.
- Two-dimensional arrays use two indices to pinpoint an individual element of the array.
- If you have an m x n array, it will have m * n elements and will require m*n*element size bytes of storage.

Example

- **int table [2] [3] = { 1,2,3,4,5,6 };**

  It means that element

  table [0][0] = 1;

  table [0][1] = 2;

  table [0][2] = 3;

  table [1][0] = 4;

  table [1][1] = 5;

  table [1][2] = 6;

Example

```
#include <iostream>
using namespace std;
int main () {
  // an array with 5 rows and 2 columns.
  int a[5][2] = { {0,0}, {1,2}, {2,4}, {3,6},{4,8}};
  // output each array element's value
  for ( int i = 0; i < 5; i++ )
    for ( int j = 0; j < 2; j++ )     {
      cout << "a[" << i << "][" << j << "]: ";
      cout << a[i][j]<< endl;
    }
  return 0;
}
```

Arrays as Parameters

- Two-dimensional arrays can be passed as parameters to a function, and they are passed by reference.
- When declaring a two-dimensional array as a formal parameter, we can omit the size of the first dimension, but not the second; that is, we must specify the number of columns.
- void print(int A[][3],int N, int M)
- In order to pass to this function an array declared as:
- int arr[4][3];
- we need to write a call like this:
- print(arr);

Example

```
#include <iostream>
using namespace std;
void print(int A[][3],int N, int M) {
  for (R = 0; R < N; R++)
    for (C = 0; C < M; C++)
       cout << A[R][C];
}
int main () {
  int arr[4][3] ={{12, 29, 11},
                {25, 25, 13},
                {24, 64, 67},
                {11, 18, 14}};
  print(arr,4,3);
  return 0;
}
```

List Data Structure

Definition

- A list is the collection of items of the same type (grocery items, integers, names).
- The data which we store in list should be of same nature.
- The items, or elements of the list, are stored in some particular order.
- Example; You may have names in some alphabetical order i.e. the names which starts with *A* should come first followed by the name starting with *B* and so on. The order will be reserved when you enter data in the list.

Operations

Operation Name	Description
createList()	Create a new list (presumably empty)
copy()	Set one list to be a copy of another
clear();	Clear a list (remove all elements)
insert(X, ?)	Insert element X at a particular position in the list
remove(?)	Remove element at some position in the list
get(?)	Get element at a given position
update(X, ?)	Replace the element at a given position with X
find(X)	Determine if the element X is in the list
length()	Returns the length of the list.

Operations

We need to know what is meant by “particular position” we have used “?” for this in the above table. There are two possibilities:

- Use the actual index of element: i.e. insert it after element 3, get element number 6. This approach is used with arrays
- Use a “current” marker or pointer to refer to a particular position in the list.

Operations

If we use the “current” marker, the following four methods would be useful

Functions	Description
start()	Moves the “current” pointer to the very first element
tail()	Moves the “current” pointer to the very last element
next()	Move the current position forward one element
back()	Move the current position backward one element

Implementation

- Suppose we want to create a list of integers. For this purpose, the methods of the list can be implemented with the use of an array inside.
- For example, the list of integers (2, 6, 8, 7, 1) can be represented in the following manner where the current position is 3.

A	2	6	8	7	1				current	size
	1	2	3	4	5				3	5

Implementation

- In this case, we start the index of the array from 1 just for simplification, which is actually the second position.

A	2	6	8	7	1			current	Size
	1	2	3	4	5			3	5

Implementation

1. **add Method**

Suppose there is a call to add an element in the list i.e. *add(9)*. As we said earlier that the current position is 3, so by adding the element 9 to the list, the new list will be (2, 6, 8, 9, 7, 1).

To add the new element (9) to the list at the current position, at first, we have to make space for this element.

Shift every element on the right of 8 (the current position) to one place on the right.

add Method

After creating the space for new element at position 4, the array can be represented as

A	2	6	8		7	1			current	size
	1	2	3	4	5				3	5

In the second step, we put the element 9 at the empty space i.e. position 4. Thus the array will attain the following shape.

A	2	6	8	9	7	1			current	size
	1	2	3	4	5	6			4	6

add Method

We have moved the current position to 4 while increasing the size to 6.

next Method

- In this method, we do not add a new element to the list but simply move the pointer one element ahead.
- This method is required while employing the list in our program and manipulating it according to the requirement.
- We have two variables- *current* and *size* to store the position of current pointer and the number of elements in the list.

next Method

- By looking on the values of these variables, we can find the state of the list i.e

  How many elements are in the list

  At what position the current pointer is
- The method *next* is used to know about the boundary conditions of the list i.e. the array being used by us to implement the list.

next Method

- For Example, 100 elements are added to the array. When we want to add 101st element to the array. We used to move the current position by *next* method and reached the 100th position. Now, in case of moving the pointer to the next position (i.e. 101st), there will be an error as the size of the array is 100, having no position after this point.

remove Method

- The *remove* method removes the element residing at the current position.
- Suppose there are 6 elements (2, 6, 8, 9, 7, 1) in the list. The current pointer is pointing to the position 5 that has the value 7. We remove the element, making the current position empty. The size of the list will become 5.

A	2	6	8	9		1			current	size
	1	2	3	4	5	6			5	~~6~~ 5

remove Method

- We fill in the blank position left by the removal of 7 by shifting the values on the right of position 5 to the left by one space.
- The current pointer remains pointing to the position 5 despite the fact that there is now element 1 at this place instead of 7.

A	2	6	8	9	1				current	size
	1	2	3	4	5				5	5

find Method

- The *find (x)* function is used to find a specific element in the array.
- We pass the element, which is to be found, as an argument to the *find* function.

Other Methods

- *get() Method* gets the element from current position in the array
- *return A[current]* returns the element to which the *current* is pointing to (i.e. the current position) in the list A.
- *update(x)* is used to change (set) the value at the current position. A value is passed to this method as an argument. It puts that value at the current position.

Other Methods

- *back()* method decreases the value of variable *current* by 1. In other words, it moves the current position one element backward.
- *start()* method sets the current position to the first element of the list. We know that the index of the array starts from 0 but we use the index 1 for the starting position.
- the *end()* method sets the current position to the last element of the list.

Analysis of Array List

- We will analyze different methods used for the implementation of the list.
- We will the level up to which these are efficient in terms of CPU's time consumption
- Following are the methods for array list analysis

Add

- When we add an element to the list, every element is moved to the right of the current position to make space for the new element.
- if the current position is the start of the list and we want to add an element in the beginning, we have to shift all the elements of the list to the right one place.
- This is the worst case of adding an element to the list.

Add

- Example

Suppose if the size of the list is 10000 or 20000, we have to do the shift operation for all of these 10000 or 20000 elements.

- if we add an element at the end of the list, it can be done by carrying out ‘no shift operation’. It is the best case of adding an element to the list

Remove

When we remove an element at the current position in the list, its space gets empty.

To fill this space, we shift the elements on the right of this empty space one place to the left.

If we remove an element from the beginning of the list, then we have to shift the entire remaining elements one place to the left.

Remove

- Example

Suppose there is a large number of elements, say 10000 or 20000, in the list. We remove the first element from the list. Now to fill this space, the remaining elements are shifted (that is a large number). Shifting such a large number of elements is time consuming process. The CPU takes time to execute the *for* loop that performs this shift operation. Thus to remove an element at the beginning of the list is the worst case of *remove* method. However it is very easy to remove an element at the end of the list.

Find

- The *find* method takes an element and traverses the list to find that element.
- The worst case of the find method is that it has to search the entire list from beginning to end. So, it finds the element at the end of the array or the element is not found.

List using Linked Memory

- In an array, the memory cells of the array are linked with each other. It means that the memory of the array is contiguous.
- It is impossible that one element of the array is located at a memory location while the other element is located somewhere far from it in the memory.
- It is not possible to increase or decrease the size of an array during the execution of the program.

List using Linked Memory

- To avoid such problems, there is need of using linked memory in which the various cells of memory, are not located continuously.
- Each cell of the memory not only contains the value of the element but also the information where the next element of the list is residing in the memory.
- It is not necessary that the next element is at the next location in the memory. It may be anywhere in the memory

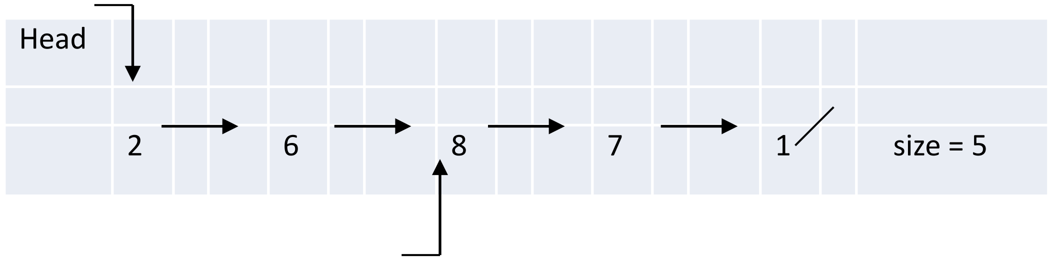
Linked List

- A linked list is a collection of objects linked together by references from one object to another object.
- By convention these objects are named as nodes.
- So the basic linked list is collection of nodes where each node contains one or more data fields AND a reference to the next node.
- The last node points to a NULL reference to indicate the end of the list.

Linked List

- *Node*

A node comprises two fields. i.e. the *object* field that holds the actual list element and the *next* that holds the starting location of the next node.



Linked List

Head

- In the linked list we need to know the starting point of the list.
- we have a pointer *head* that points to the first node of the list.
- If we don’t use *head*, it will not be possible to know the starting position of the list

Linked List

Current

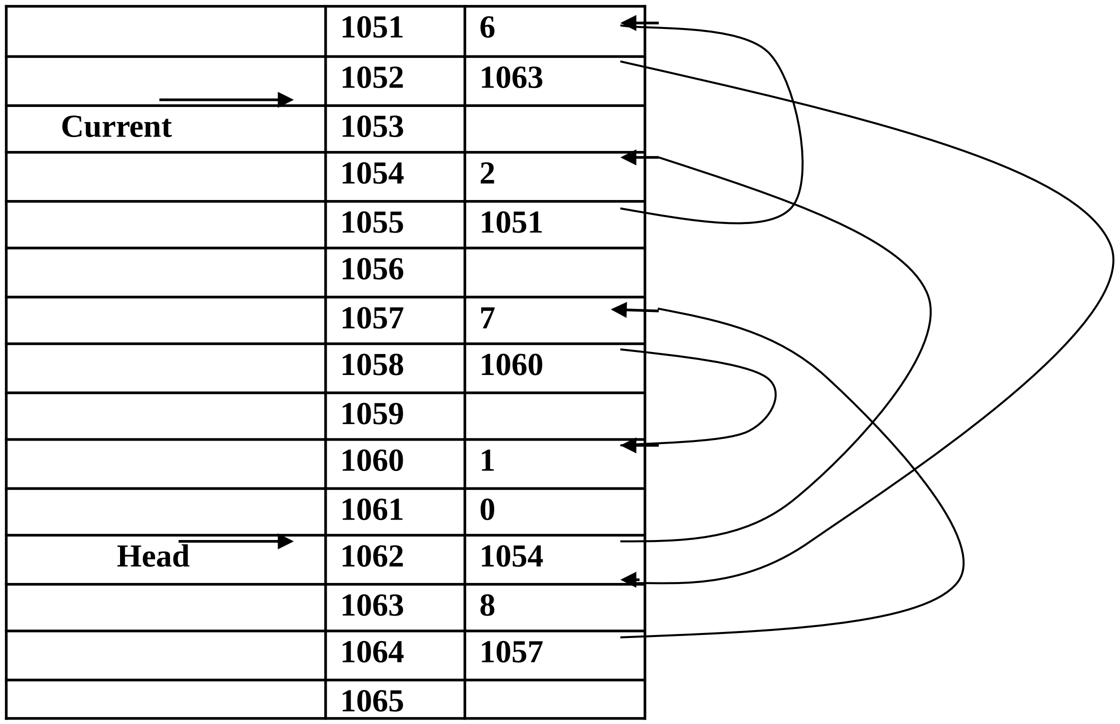
- A pointer *current* points to the current node of the list.
- We need this pointer to add or remove current node from the list.

Null

- We place the memory address NULL in the last node.
- NULL is an invalid address and is inaccessible.

Linked List

The following diagram depicts the process through which this linked list is stored in the memory.



	1051	6
	1052	1063
Current	1053	
	1054	2
	1055	1051
	1056	
	1057	7
	1058	1060
	1059	
	1060	1
	1061	0
Head	1062	1054
	1063	8
	1064	1057
	1065	

Node Designing

- A node is a struct with at least a data field and a reference to a node of the same type.
- A node is called a self-referential object, since it contains a pointer to a variable that refers to a variable of the same type.

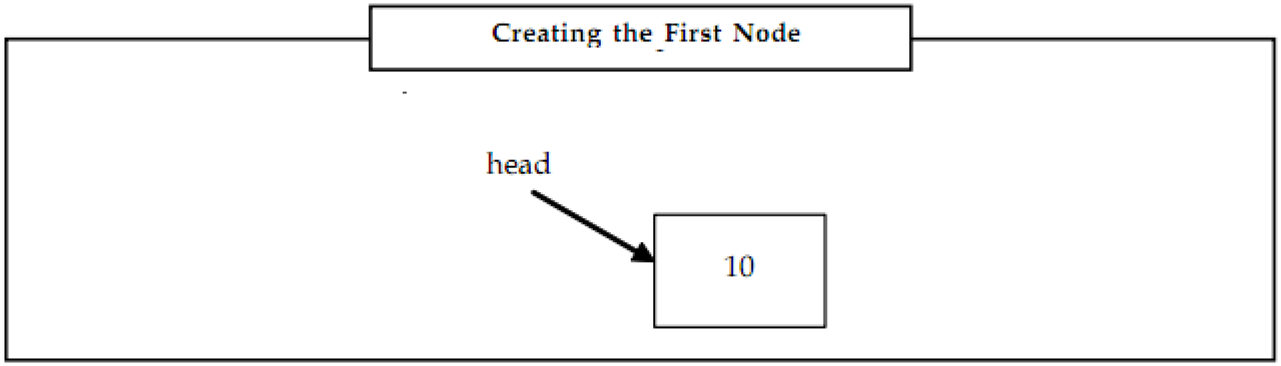
Node Designing

Example: A struct Node that contains an int data field and a pointer to another node can be defined as follows.

```
typedef struct node {
 int data;
 struct node* next;
} node;
node* head = NULL;
```

Creating First Node

- Memory must be allocated for one node and assigned to head as follows.
- head = (node*) malloc(sizeof(node));
- head→data = 10;
- head→next = NULL;



Adding Second Node and Linking

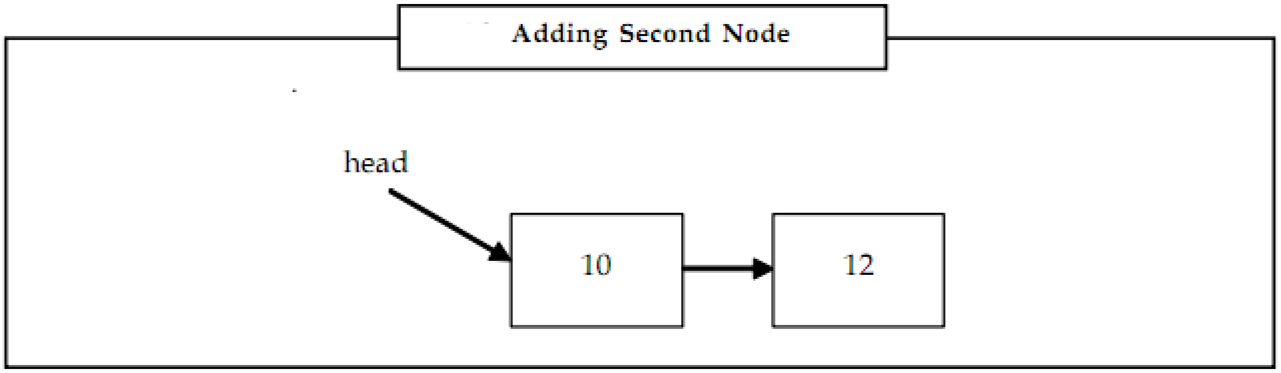
- The second node is added in the following way:

node nextnode = malloc(sizeof(node));*

nextnode → data = 12;

nextnode → next = NULL;

head → next = nextnode;



Example 1

```
#include <iostream>
using namespace std;
struct list
{
			int a[6];
			void createlist (int c[6])
			{
						for (int i=0; i<=5; i++)
						{			cin>>a[i];
						c[i]=c[i];}
			}
			void show()
			{
						for (int i= 0; i<=5; i++)
						{cout<<"the value at position "<<i<<" is : ";
						cout<<a[i]<<endl;}}
			void insert()
			{
						int b;
						cout<<"enter the position to insert from 0- 5: ";
						cin>>b;
						cout<<"enter the value to insert : ";
						cin>>a[b];
			}
			void find ()
			{
						int s;
						cout<<" enter a position to find the value from 0-5 ";
						cin>>s;
						cout<<" the value for a given number is : ";
						cout<<a[s];}
```

•

Example 1

```
void update()
                {
                                int y;
                                cout<<" enter a position to update : ";
                                cin>>y;
                                cout<<" enter a value to update ";
                cin>>a[y];
                                cout<<" the update number is "<<a[y]<<" from ";
                                                }
                void get ()
                {
                                int w;
                                cout<<"enter the position to get the value of it from 0-5 : ";
                                                cin>>w;
                                                cout<<" value at position no "<<w<<" is after insertion is "<<a[w];
                }
                void clear()
                {
                                int i;
                                cout<<" just clear all the list so "<<endl;
                                for ( i=0; i<6; i++)
                                                {a[i]=0;        }};
```

Example 1

```
void main ()
{
                    cout<<"  NAME   : ALi  \n\n SUBJECT : DATA STRUCTURE \n\n\n";
                    int q[6];
                    list list1;
                    list1.createlist(q);
                    cout<<endl;
                    list1.show();
                    cout<<endl<<endl;
                    list1.insert();
                    cout<<endl<<endl;
                    list1.show();
                    cout<<endl<<endl;
                    list1.find();
                    cout<<endl<<endl;
                    list1.update();
                    cout<<endl<<endl<<endl;
                    list1.get();
                    cout<<endl<<endl;
                    list1.clear();
                    cout<<endl<<endl;
                    list1.show();
cout<<endl<<endl;
```

- }
-

Example 2

```
•     #include<stdio.h>
•     #include<conio.h>
•     struct single_link_list
•     {
•       int age;
•       struct single_link_list *next;
•     };
•     typedef struct single_link_list node;
•     node *makenode(int );
•     int main()
•     {
•       int ag;
•       node *start,*last,*nn;   //nn=new node
•       start=NULL;
•       while(1)
•       {
•          printf("Enter your age : ");
•          scanf("%d",&ag);
•          if(ag==0)
•             break;
•          nn=makenode(ag);
•          if(start==NULL)
```

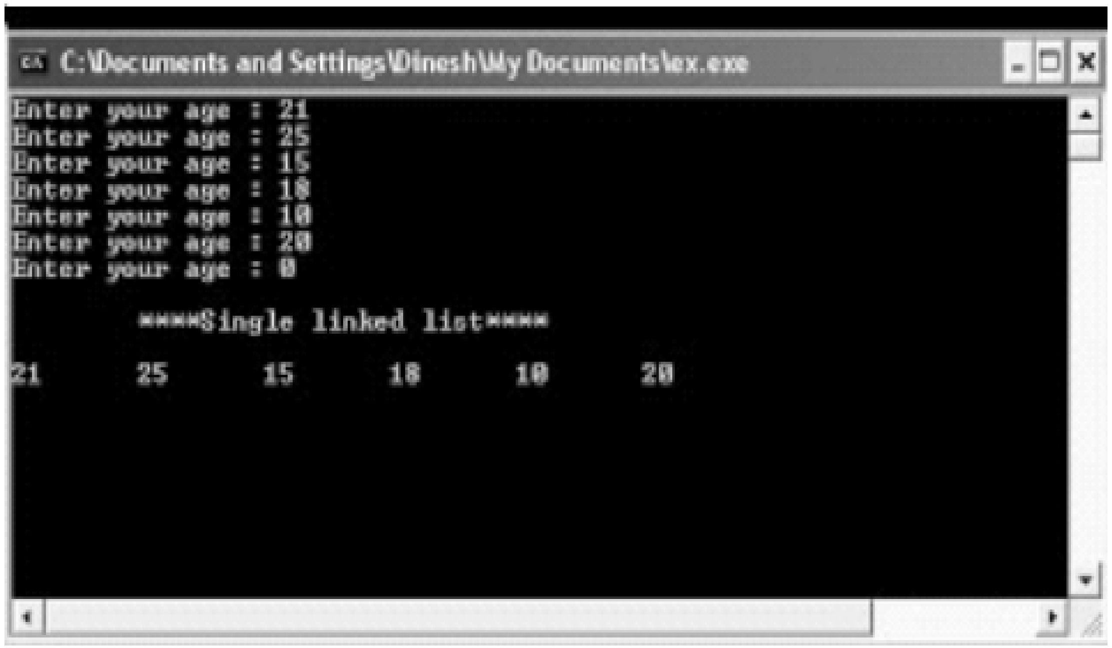
Example 2

```
•   /*creation of node*/
•   node *makenode(int tmp)
•   {
•    start = nn;
•           last = nn;
•        }
•        else
•        {
•           last->next = nn;
•           last = nn;
•        }
•     }
•     printf(“\n\t****Single linked list****\n\n”);
•     for(; start!=NULL; start=start->next)
•        printf(“%d\t”,start->age);
•     getch();
•     return 0;
•   }
•   /*creation of node*/
•   node *makenode(int tmp)
•   {
```

Example 2

```
•  {
•  start = nn;
•        last = nn;
•     }
•     else
•     {
•        last->next = nn;
•        last = nn;
•     }
•   }
•   printf(“\n\t****Single linked list****\n\n”);
•   for(; start!=NULL; start=start->next)
•      printf(“%d\t”,start->age);
•   getch();
•   return 0;
• }
```

Example 2



```
Enter your age : 21
Enter your age : 25
Enter your age : 15
Enter your age : 18
Enter your age : 10
Enter your age : 20
Enter your age : 0

        ****Single linked list****

21      25      15      18      10      20
```