

Programming and Data Structures
Prof. N. S. Narayanaswamy
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

Lecture – 08
Doubly Linked Lists and Arrays

Welcome to lecture number 9 of the course on programming and data structures. So, in the last lecture this week, we looked at comparing the implementations of link list with doubly linked list. In the way we did that is we took a single problem and in this case, it was a big integer addition and we looked at the advantages of maintaining the number in a link list over maintaining the number in a doubly linked list over maintaining it in a singly linked list.

The way we did that was we ensure that our program not only perform the arithmetic operations, but also presented some statistics about how many times pointers were traversed when the data was maintained as a link list and when the data is maintained as a doubly link list. This gave us a experimental way of comparing the efficiency of two implementations.

One of the things that we did not do in the last lecture was we did not look at how much memory was used and we will definitely do that in the course of a few lectures. But today what we are going to do is, we are going to compare a link list implementation against an implementation with an array. So, essentially at the end of this week, we would have seen three data types: linked list, doubly linked list and arrays, and we will have a clear understanding of what purpose, what data structure, what or what data types serves.

And again what we will do is, we will look at one particular problem and we will write two programs or we essentially write one program with two different functions, one solving the problem using the doubly linked list data type, the other solving the problem using the array data type. And we will print out statistics of which will be a representative of the efficiency of the two implementations that will help us come up with the concrete comparison.

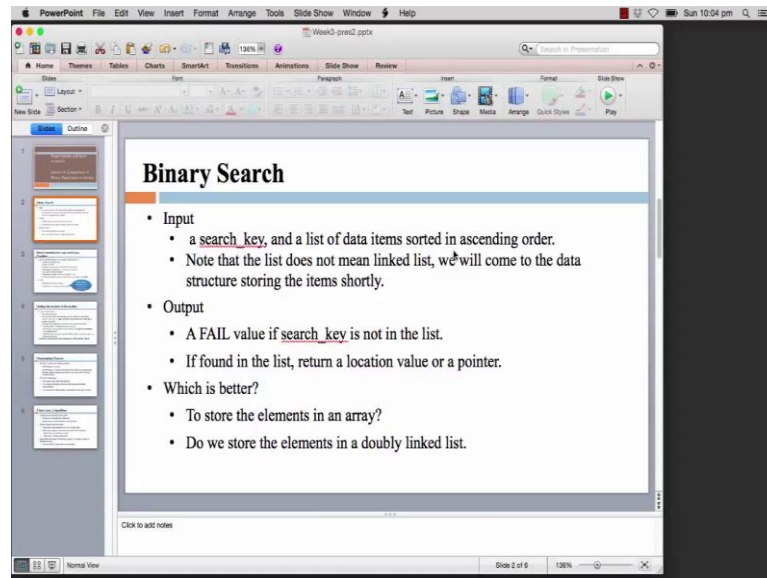
The problem and question for today's lecture is the well known binary search algorithm. We call that and if you do not know, then it is a very easy problem, it is a very interesting programming exercise. You have sorted data and you can assume with the data sorted in ascending order and the question is that, you are given a key and you have to check, if the key is present in the sorted data set or not.

The approach is very simple, you will kind of check the median element and check, if the given query item is the median element or not or whether, it is smaller than the median element or the larger than the median element. Depending upon the comparison result that is whether it is smaller than the median element or larger than the median element, we shrink the size of the search space; the list size shrinks by a factor of 2, so because you are comparing with the median element.

This is the idea of binary search and it is often used as an introduction to efficient search procedures and you might have heard or studied that kind of log and comparisons, where n is a total number of keys which are there are required by binary search. But today what we are going to do is, we are going to explore the relevance of the underlying data structure. Remember that I said n sorted keys are available and I did not tell you, whether they have present in a linked list or whether they have present in an array.

And what we are going to do today is, we are going to compare and contrast the two implementations with respect to efficiency. In particular, we will come up with an estimate of which one is more efficient than the other. And that will essentially be the last lecture for this week and we will also look at the implementation of these two programming exercises in today's lecture and talk about what we will be doing in the next week. So, now I will just go to the slides.

(Refer Slide Time: 04:34)



So, comparison of binary search lists versus array that is the focus of today's lecture. So, let us just look at binary search, we have already discussed it. The input binary search is a search key and a list of data items stored in ascending order. Now, when I mean list here, I do not mean a linked list, we will come to the data structure storing the items in a couple of minutes. The output of the binary search procedure is a fail value, you can either return minus 1 or some special symbols saying the research has failed.

If a search key is not in a list, if it is found in the list our goal is to return a location value, in other words a pointer to the particular location that contains that value. So, also assume that the data items are all distinct data items, there are no repetitions. So, which is better? So, do we store the elements in an array or do we store the elements in a doubly linked list? So, observe that the doubly linked list is better than storing in a linked list with respect to time, because a number of pointer accesses as we have seen are reduced, when we move from a singly linked list to a doubly linked list.

We of course use more space, when we implemented doubly linked list. So, how to quantify the space, we have not yet done, but as a qualitative statement one can always say that one uses the little bit more space as such it is shown one more pointer per node. So, today's lecture as I said earlier is a comparison between these two, do we store the elements in an array or do we store the elements in a doubly linked list.

(Refer Slide Time: 06:27)

Binary Search(list,start, end, search_key) - Procedure

- Take the middle element, call it median, in the sorted list.
 - Compare with Search_key.
 - If equal, FOUND.
 - If median is larger, then search to the left of median
 - **Recursively search** Start – predecessor of median
 - Else search to right of median
 - **Recursively search** Successor of median - end
 - If start and end become same and search_key not there, report **fail**
- Effort
 - Finding location of the median
 - Comparison of search_key with median

Search range
shrinks by $\frac{1}{2}$, so
this cannot be
changed much.

So, let us just become familiar with the binary search procedure and more importantly the focus of this slide is to show you the arguments to the binary search function. The first one is the list itself. The second one is the starting index in the list which is important and the final index of the list and the value of the search key. Let us understand the arguments, the question here is in the list starting with the index start and ending with the index end, is the search key present between these two indices. This is the goal of the binary search function.

To answer yes or no, I am return a pointer to the position where the key is present, I hope this is clear. So, now what we do is we take the middle element, let us say this is the median, let us call it the median in the sorted list between the indices start and end. Now, we compare the median element with the search key, if it is equal to the search key, then you have founded, on the other hand, if the median is larger, then we recursively search to the left of the median.

What do I mean by the left of the median? We search in the region start index to the predecessor position of the median for search key, otherwise we search to the right of the median that is we look at the successor of median up to the end of the search region for the search key. Now of course, it is very important for us to say, when we will report that

is search has failed to find the key. If start and end become the same, then this and if search key is not in that particular location, then we report failure.

The other way of thinking of it is that when the start and end cross each other and you have still not found the search key, where it means that the search key is not available in the given data set. It is a small exercise to formally write a mathematical argument, it supports it is. So, if binary search reaches the situation, where the start and the end values are become equal or they have crossed, meaning the start index has become more than the end index and the key is not yet found, then you can safely asset that the key is not there in the given sorted list.

So, this is the algorithm it is a fairly straight forward and very nicely specified recursive algorithm and we are going to today analyze whether storing it in the list or storing it in a doubly linked list or storing it inside an array is better. Now, let us to understand the exercise, understand and design the appropriate data structures, let us understand where the computational effort is. The computational effort is in finding the location of the median.

Now, that is the most important thing, because you need to take the middle element and therefore, we need to find the location of the median that is you take the sorted list and pick the median. Now, the second thing is that you need to compare the search key with the median value, but observe the total number of comparisons is only log of the size of the log of the total number of elements in your list; it is the logarithm of logarithm base 2 of the total number of elements in the list.

The reason is what is here in this blue color column, in every step the search range shrinks by a factor of 2. In other words, if you started off with say 32 elements after the first comparison with the median element, the research range will shrink to the value 16, then it will shrink to 8, then to 4, then to 2, then to 1. As you can see 2 power 5 is 32 and we have performed only five comparisons log 32 is equal to 5. Now, we believe that this cannot be significantly improved. That is the number of comparisons that you performed by binary search cannot become smaller than $\log n$, where there are n elements in the list.

However, we will see how efficient we can make this problem of finding the median element in the list in the sorted list.

(Refer Slide Time: 11:04)

Finding the location of the median

- If size of list is known
 - Then store in an array.
 - Since keys are sorted in ascending order, the median is in the middle location of the array- **a single arithmetic operation $(start+end)/2$ gives location of median**
- If elements are arranged in a linked list, even a doubly linked list
 - Find the number of elements in the list, say size.
 - Then traverse $size/2$ pointers to get a pointer to the **position containing the median element**.
 - Multiple pointer accesses to get the middle position- much more than one arithmetic operation
- **So using an array is better when compared to a list for binary search**

So, the goal is the following, so we started off with binary search and we now localized and come to the most important question, we want to find the location of the median. So, how does one find the location of the median? Now, let us consider two cases, where let us consider the case, let us assume that we know the size of the list and then, consider two cases.

Now, if the size of the list is known and if he store the sorted keys in an array of that particular size and they are sorted in ascending order, then the median is right in the middle location of the array and a single arithmetic operation, which is the start index plus an end index divided by 2 gives the location of the median. Give a small exercise that each of you should complete on a sheet of paper, check that if you want to find the middle element, it is the start index plus end index divided by 2.

Now, if the elements are arranged in a linked list, let us assume that they are arranged in doubly linked list. First, we need to find the total number of elements in the list, which is the size that we have to traverse size by 2 pointers to get a pointer to the position

containing the median element, I hope this is clear. So, if you have like 32 elements in the list and the median element is the 16th element, so you must traverse 16 pointers starting from the start pointer to get to the list, get to the position containing the median element.

Therefore, each pointer x is considered as one single operation and one can assume that this single operation is same as in time to a single arithmetic operation. In other words, you can think of this as a single CPU instruction, one arithmetic operation compared with one pointer access operation. So, clearly to get to the middle element of the array only one arithmetic operation is required, no matter how last the array is...

On the other hand, if you store the data in a list in ascending order, then to get to the median element, you have to use the size of the array divided by two numbers or divided by two pointer traversals to get to the median element. So, therefore using an array is better when compare to the list for binary search. Now, this is a qualitative statement, what we will do in our program is to write the code in a very careful fashion and run it and we will see the output and it will give us a proof that our qualitative interpretation is indeed, justified.

Of course, one can do a mathematical proof, which is what is done in a formal data structures course and I encourage you to look at the material and of formal data structures course. But today what we will do is we write a program and we will come up with an experimental estimate that storing the data in an array is better for binary search.

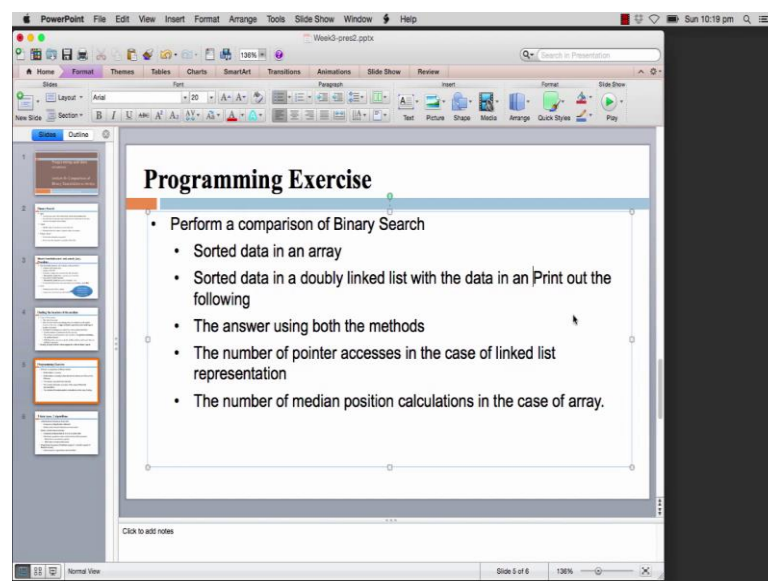
(Refer Slide Time: 14:19)

Programming Exercise

- Perform a comparison of Binary Search
 - Sorted data in an array
 - Sorted data in a doubly linked list with the data in an arraythe Big Number addition again using both the List type and the Doubly Linked List type.
- Print out the following
 - The answer using both the methods
 - The number of pointer accesses in the case of linked list representation
 - The number of median position calculations in the case of array.

So, this is the programming exercise for which we are going to take a look at the code. We want to compare a performance, compare binary search when implemented using data stored in array, sorted data store in an array and when the sorted data is in a doubly linked list. The output, we just make a small correction.

(Refer Slide Time: 14:53)



So, when the sorted data is stored in a doubly linked list, so what we will do is, we will print out the following. We will print out the answer using both the methods. So, in other words, we will print out, whether the given search key is been found or not, we will also print out the total number of pointer accesses in the case of a linked list implementation and we will print out the number of median position calculations in the case of an array and we will see, how much effort is involved in both of them.

So, now let us go to the program, so I hope the focus of the exercise today is clear to you, we have two data types, the array data type and the doubly linked list data type, we want to come up with the comparison of the two of them. So, we choose a programming exercise the well studied binary search algorithm to search for a key in a sorted list and we are going to implement binary search using the two data structures.

In one implementation, we will use the doubly linked list, in the second one, we will use the array and we will print out statistics of how many arithmetic operations were made. In the array implementation, we will print out how many pointer accesses were made when we use the data organize in a doubly linked list. So, now let us look at the program.

(Refer Slide Time: 16:35)

```

DLListMethods.c      Week2-pres2.pptx    week1-prog1.c
DLListMethods.o      Week2-pres3.pptx    week1-prog2
Lecture 1.pptx        Week2-rec3.cproj    week1-prog2.c
List-Interface.h      Week3-lec1.mp4      week1-prog3
List.h                Week3-pres1.pptx     week1-prog3.c
ListMethods.c         Week3-rec1.cproj     week1-rec2.cproj
ListMethods.o         abstract-data-type.pdf week1-rec3.cproj
MOOC_Syllabus_template.doc brute_force_string_search(1).txt week2-lec1.mp4
MOOC_Syllabus_template.odt brute_force_string_search.txt week2-lec2.mp4
MOOC_Syllabus_template.pdf fwdmoooc              week2-rec1.cproj
PDS-MOOC_Syllabus.pdf Intro-vid.mp4          week2-rec2.cproj
PerfDLL               gcc guidelines.docx
PerfDLL.c             pointers(1).txt

NSNarayanawamy-MacBook-Air:PD5-mooc narayanawamy$ vi BinSchComp.c
NSNarayanawamy-MacBook-Air:PD5-mooc narayanawamy$ vi pointers(1).txt
-bash: syntax error near unexpected token '('
NSNarayanawamy-MacBook-Air:PD5-mooc narayanawamy$ vi pointers(1).txt
-bash: syntax error near unexpected token '('
NSNarayanawamy-MacBook-Air:PD5-mooc narayanawamy$ vi DLList-Interface.h
NSNarayanawamy-MacBook-Air:PD5-mooc narayanawamy$ vi DLListMethods.c
NSNarayanawamy-MacBook-Air:PD5-mooc narayanawamy$ gcc -c DLListMethods.c
DLListMethods.c:46:1: warning: /* within block comment [-Wcomment]
/* count reset to zero here, for future function calls to work correctly */
^
1 warning generated.
NSNarayanawamy-MacBook-Air:PD5-mooc narayanawamy$ lvi
vi DLListMethods.c
NSNarayanawamy-MacBook-Air:PD5-mooc narayanawamy$ lgcc
gcc -c DLListMethods.c
DLListMethods.c:46:1: warning: /* within block comment [-Wcomment]
/* count reset to zero here, for future function calls to work correctly */
^
1 warning generated.
NSNarayanawamy-MacBook-Air:PD5-mooc narayanawamy$ lvi
vi DLListMethods.c
NSNarayanawamy-MacBook-Air:PD5-mooc narayanawamy$ lgcc
gcc -c DLListMethods.c
NSNarayanawamy-MacBook-Air:PD5-mooc narayanawamy$ gcc BinSchComp.c -o BinSchComp DLListMethods.o
NSNarayanawamy-MacBook-Air:PD5-mooc narayanawamy$

```

So, the comparison program is this C file, it is called BinSchComp dot c standing for

binary search comparison dot c that is the C program and let us look at the interface files, I am using a doubly linked list implementation. So, let us look at the interface files, which would be interface dot h and let us look at the methods.

(Refer Slide Time: 17:16)

```
/* DLLInterface.h created with all the function prototypes */
/* Created by M.S. Marayanasamy */
/* This file is obtained by re-using hte List-Interface.h by carefully renaming of the
methods */

#include "DLList.h"

extern int finding_mid_binSearch;
/* returns the number of elements in the list */
int Dsize( DL_node *, DL_node * );
int DisEmpty(DL_node * );

/* Decision questions given a pointer */
int DisFirst(DL_node *, DL_node * );
int DisLast(DL_node *, DL_node * );

/* function to return pointers to specific queries */
DL_node * Dfirst(DL_node * );
DL_node * Dlast(DL_node * );
DL_node * Dbefore(DL_node *, DL_node * );
DL_node * Dafter(DL_node *, DL_node * );

/* Update methods */
void DreplaceElement(DL_node *, int );
void DswapElement(DL_node *, DL_node * );
DL_node * DinsertBefore(DL_node *, DL_node *, int );
void DinsertAfter(DL_node *, DL_node *, int );
DL_node * DinsertFirst(DL_node *, int );
void DinsertLast( DL_node *, int );
DL_node * Deliminate(DL_node *, DL_node * );

/* Did you notice that there is no function to create a list. You can do */
DL_node * BinSearch(DL_node *, DL_node *, int );
~
"DLList-Interface.h" 37L, 1157C
```

So, we have the same set of methods to compute the size, to check if a list is empty and so on and so forth. These were the appropriate methods from the last lecture, but now there is one more method, which is called bin search, it returns a pointer to doubly linked list known, it takes as argument two pointers and it takes an integer query. We will talk about the meaning of the arguments in a short while, so that is a new function that I have add it.

There is one more function, one more data item that I have add it, which is the integer which keeps track of the number of pointer accesses. It is called finding mid by binary search. So, this is an integer, now I am declaring it to be an external integer, this will be use by programmers whoever uses the interface dot h file will find this integer variable declaration here and the definition of this variable and the way, it is modified is present in the doubly linked list methods dot c file, we are going to see that in short.

So, compare to the earlier implementation, this is the more current implementation,

additional methods have been added and additional data items have been added. In particular, I have added a method for the binary search and I have also kept track of one variable, added a variable to keep track of the total number of pointer accesses that are made by binary search. So, let us go to the doubly linked list methods dot c.

(Refer Slide Time: 19:04)

```
/* Extends ListMethods.c */
#include "DLList.h"
#include <stdio.h>
#include <stdlib.h>

int DL_Ptr_Travs_Count=0;
/* Variable global to this module return the num of pointers traversed in the list */
/* returns the number of elements in the list */
/* note the usage of recursion */

int finding_mid_binSearch=0;
/* This counter keeps track of times BinSearch has to find mid. For each mid query, the cost is
the number of pointer traversals to find mid. */
DL_node * DL_node = DLnode * list)
{
    if (list == NULL) return list;
    while(list->next != NULL)
    {
        list = list->next;
        DL_Ptr_Travs_Count++;
    }

    /* skip till the last DL_node is reached and then return it */
    return list;
}

int Dsize( DL_node * head, DL_node * tail)
{
    static int count=0;
    /* C programming feature that keeps count value from previous function calls*/
    /* NOTE IN TODAY'S LECTURES THAT THEY MUST BE CAREFULLY RESET */
    int ret_value;

    if (tail == NULL || head == NULL) return 0;
    if (head == tail)
    {
        ret_value = count + 1;
        count = 0;
    }
    /* count reset to zero here, for future function calls to work correctly */
    return ret_value;
}
"DLListMethods.c" 380L, 8217C
```

So, let us see, now define this counter variable finding mid bin search, we initialized it to zero and as a remark below says this counter keeps track of the number of times binary search has to find mid. For each mid query, the count is the total number of pointer traversals to find mid. I hope this is clear that is if you want to find a mid, you have to do a certain number of pointer traversals. For each mid query, we count the total number of pointer traversals and submit up over all mid queries. That is the total amount of work that is done by binary search to find the mid element.

(Refer Slide Time: 20:15)

```
DL_node * Dlast(DL_node * list)
{
    if (list == NULL) return list;
    while(list->next != NULL)
    {
        list = list->next;
        DL_Ptr_Travs_Count++;
    }

    /* skip till the last DL_node is reached and then return it */
    return list;
}

int Dsize( DL_node * head, DL_node * tail)
{
    static int count=0;
    /* C programming feature that keeps count value form previous function calls*/
    /* NOTE IN TODAY'S LECTURES THAT THEY MUST BE CAREFULLY RESET */
    int ret_value;

    if (tail == NULL || head == NULL) return 0;
    if (head == tail)
    {
        ret_value = count + 1;
        count = 0;
        /* count reset to zero here, for future function calls to work correctly */
        return ret_value;
    }

    /* meaningless call return -1*/
    DL_Ptr_Travs_Count++;
    if (head->next == tail)
    {
        ret_value = count+2;
        /* Make a copy of ret_value */
        count = 0;
        /* count reset to zero here, for future function calls to work correctly */
        return ret_value;
    }

    count=count+2;
    return Dsize(head->next,tail->prev);
}
```

We have also modified the Dsize function that is a size function which returns a size of the list, so this function is where the... This function is used to compute the size of the list of a doubly linked list between the nodes pointed to by head and tail.

(Refer Slide Time: 20:35)

```
/*Did you notice that there is no function to create a list. You can do
that by a function when you want to */

/* This is very crucial function used to come up with the stopping conditions for Binary Search in a list */
int comes_after(DL_node *start, DL_node *end)
{
    if (start == end) return 1;
    if (start == NULL) return 0;
    return comes_after(start->next,end);
}

/* Function to find the middle position in a list whose first and last is given as arguments */
DL_node * midpoint(DL_node * start, DL_node * end)
{
    DL_node * mid;
    int center;
    int loop_counter;

    if ( !comes_after(start,end) ) return NULL;
    /* Meaning less function calls are caught here */

    center = Dsize(start,end)/2;
    /* center now contains the number of pointer traversals to get to the middle element */

    finding_mid_binSearch = center + finding_mid_binSearch;
    /* This is the counter which keeps track of number of pointerse for finding middle */

    mid = start;
    for (loop_counter=0; loop_counter < center; loop_counter++)
    {
        if (mid == NULL) break;
        mid = mid->next;
    }

    /* Here mid is pointing to the middle elements */
    return mid;
}
```

So, let us just look at this scope very quickly, so let us just look at the recursive function

call. Recursive function call shrinks the list from left and right that is the pointer moves to mid pointer of next and tail pointed at previous. Here are the boundary conditions, if head is equal to tail, then you increment the value of count and return that value. Observe something very important here. Here I have reset count to zero and return the value return value.

This is because count is a static integer variable, it must be return set to zero, it must be reset to zero for future function calls to work correctly. This is very important, so please pay attention. If count was not reset to zero, subsequent function calls to Dsize may not give you the correct answer, so as a good programming practice when you use static int with recursion, you must be very careful about the counter variable which is a static variable.

So, when you use a static variable inside a recursive function call, you must be clear about what the role of that variable is. Here, because I am repeatedly going to make function calls to Dsize to identify the size of the list. After one computation of the size of the list, I reset count to zero and return a different, return the value of a different container or a different variable. If head pointer of next is equal to tail, then we are essentially counting two new elements, two new locations therefore, count gets increased by two, we gets reset and then I return the value.

Otherwise, what is otherwise mean that head is not equal to tail, head pointer of next is not equal to tail which means head and tail are at least, between the two of them there is at least one more node. We increase the value of count by 2 and then make a recursive function call by moving the head pointer forward and the tail pointer backwards. That is head now points to head pointer of next and tail points to tail pointer of previous. Observe that here in this implementation we are very carefully using features of the doubly linked list. Rest of the piece of the code or doubly linked list methods which are unchanged, let us go directly to the binary search implementation.

(Refer Slide Time: 23:03)

```
int center;
int loop_counter;

if ( !comes_after(start,end) ) return NULL;
/* Meaning less function calls are caught here */

center = Dsize(start,end)/2;
/* center now contains the number of pointer traversals to get to the middle element */

finding_mid_binSearch = center + finding_mid_binSearch;
/* This is the counter which keeps track of number of pointerse for finding middle */

mid = start;
for (loop_counter=0; loop_counter < center; loop_counter++)
{
    if (mid == NULL) break;
    mid = mid->next;
}

/* Here mid is pointing to the middle elements */
return mid;
}

/* Binary Search on doubly linked list that returns a pointer to the position containing the key or
returns NULL */
DL_node * BinSearch(DL_node * start, DL_node * end , int query)
{
    DL_node * mid;

    if (start == NULL || end == NULL) return NULL;
    if ( !comes_after(start,end) ) return NULL;
    /* The above to return conditions for the recursion */

    mid = midpoint(start,end);
    if (mid == NULL) return NULL;
    /* Not a very meaningful condition, but just checking so that the
    next check is mid->value, and it must work meaningfully. A bit of
    DEFENSIVE programming */
```

So, let us just look at it, binary search is a name of the function, it returns a pointer to a node in the doubly linked list. It takes a pointer to a node called stack and takes a pointer to another node called end and it takes in the query value. Mid is going to be the return value of the point which is going to contain the key if it is there, otherwise we will return null. So, let us look at some boundary conditions, if start becomes null or end becomes null, we return null.

This condition is a come after function which is a method internal to this module, I repeat comes after is a method that is internal to the module by which I mean that it is not present in the interface files. So, comes after of start and end, the role of the function is very clear, we check if end comes after start. So, if an end comes before start then we return null saying with the key is not found. So, comes after is a very simple function, it is just a two line piece of code.

Let us take a look at the come after function, it is just three lines ((Refer Time: 24:33)). Comes after takes two pointers in a list, if start is equal to end it says 1, if start is equal to null it return 0, otherwise it moves the starts pointer forward and it goes and tries to meet end, so it is a recursive function again. So, as you can see how recursion places a very important role in writing programs to maintain data structures.

So, we have so many recursive functions we have seen so far. So, here is a come after which is a recursive function, we checks if the end pointer comes after the start point. So, now let us go back, so comes after basically returns 1, if end comes after start. This one says that if end does not come after start, then you return a null. Now, if you have not return, if the control has not return from this function call and it comes here, it means that start and end that is end comes after start, then neither start and end are null.

So, now what you do is you compute the midpoint using the midpoint function. What is midpoint? If the midpoint function essentially returns a pointer to a location which is the median element in the list, where the first element is the start and the last element is end. So, let us just look at this midpoint is the function, we will come to the midpoint in one minute, let us complete the binary search. So, mid returns, mid is essentially a pointer to the middle point of the list. For some reason if mid is equal to null, then you return null.

Observe that this is an example of what I would call or what is known as defensive programming. It is not a very meaningful condition, but I am putting this as a safety check saying that I will return null, because in the next statement I am checking if query is mid pointer of value and this is very important because, if indeed mid turned out to be null for some reason, of course the logic if you make meaningful function calls, then mid will never be equal to null, but then if meaningless function calls have been made along the way we do not know how many ways meaningless function calls can be made.

This is the safety check to ensure that the program actually exists meaningfully or return some meaningful pointers in this case it returns a null pointer. And again I asset that the reason that this comparison is there is because, in a next statement I am going to check mid pointer dot value and control will come to this particular statement, only if mid is not equal to null. You can very well imagine that if mid was null and I did not make this check, then mid pointer dot value would be a meaningless pointer access which can cause the program to actually crash.

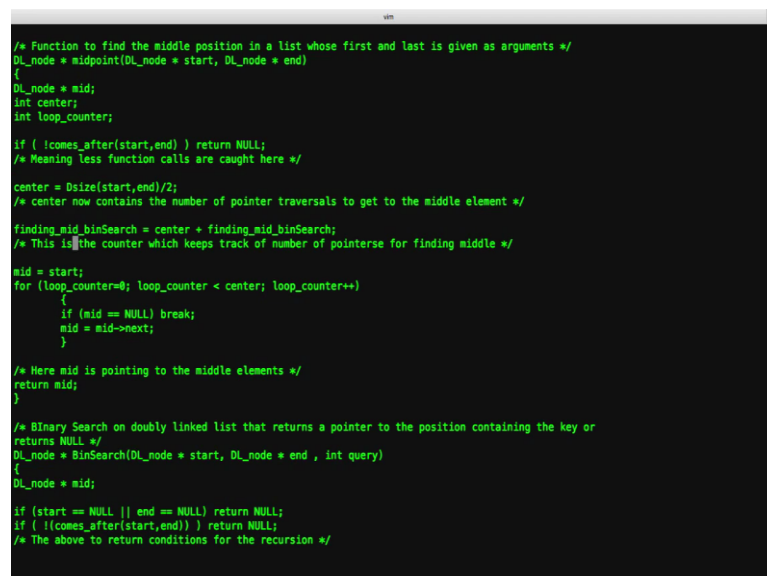
Therefore, as a programmer you must program in such a way that your program does not crash. You may not be able to handle all errors actually you know, you will not be able to handle all kinds of an errors. However, you must design your programs, so that you can

exist gracefully, a crashing program is not a good program. And therefore, this check is a kind of a defensive programming to ensure that the program does not crash here.

So, now here I check if query is equal to the value inside the mid pointer in that particular location, if it is when you found the queried value return the pointer. Otherwise, we set up the recursion. If query is more than the mid value, then we make a recursive binary search, recursive call to binary search starting with the successor of mid right up to the end and the same query value. Otherwise, it means the query is strictly smaller than the mid pointer of value, then you recurs a binary, cause a recursive binary search from start to mid pointer dot previous and the queried value.

So, observe that we using the properties of that doubly link list here very carefully. All these are a very simple, single pointer queries. If this was not a doubly link list then computing the previous as you know was a very time consuming task, multiple pointer traversals have to be made. So, this is the complete implementation of binary search, assuming that the data is arranged from a doubly link list. Let us look at the midpoint function.

(Refer Slide Time: 29:20)



```
/* Function to find the middle position in a list whose first and last is given as arguments */
DL_node * midpoint(DL_node * start, DL_node * end)
{
    DL_node * mid;
    int center;
    int loop_counter;

    if ( !comes_after(start,end) ) return NULL;
    /* Meaning less function calls are caught here */

    center = Dsize(start,end)/2;
    /* center now contains the number of pointer traversals to get to the middle element */

    finding_mid_binSearch = center + finding_mid_binSearch;
    /* This is the counter which keeps track of number of pointerse for finding middle */

    mid = start;
    for (loop_counter=0; loop_counter < center; loop_counter++)
    {
        if (mid == NULL) break;
        mid = mid->next;
    }

    /* Here mid is pointing to the middle elements */
    return mid;
}

/* Binary Search on doubly Linked list that returns a pointer to the position containing the key or
returns NULL */
DL_node * BinSearch(DL_node * start, DL_node * end , int query)
{
    DL_node * mid;

    if (start == NULL || end == NULL) return NULL;
    if ( !comes_after(start,end) ) return NULL;
    /* The above to return conditions for the recursion */
}
```

So, if you look at the midpoint function, again it takes two pointers to two locations in

doubly link list and it returns the midpoint, but along the way I also do a little bit more of work here. I also compute how many pointer traversals were made to compute the midpoint. See this, finding mid bin search. Remember this is the counter variable to keep track of how many pointer traversals are made totally.

This is the variable global to this module and center tells you that the number of pointers traversals that you will have to make is the size of the list divided by 2 that gets added to the previous value of a finding mid bin search and that is it. Everything is fairly straight forward, here you look right up to the center of the list and keeps skipping to the next location till you come to the midpoint. Again here is a bit of defensive programming, if mid becomes null for some reason which is exits from the loop and return mid is null, again this is a bit of defensive programming.

So, that completes the whole discussion about the implementation of the few additional methods, let we just draw your attention to how many additional methods I have implemented. So, the bin search is a method, mid point is a method and comes after is a method, so I have implemented three more methods in the doubly link list data type methods file. But observe that only the last function is in the interface file.

Therefore, if you are using this data type right, you will not you are not supposed to use function calls to the midpoint method and the comes after method. These methods are considered to be methods private to this module. I will not see more about it, I have just introduce the very important keyword, you are not supposed to make function calls to these methods which have not being put in the interface file. Of course, because they are not put in an interface file, if you try to use them, the compilation will fail. It will give you a lot of error messages.

(Refer Slide Time: 31:56)

```
finding_mid_binSearch = center + finding_mid_binSearch;
/* This is the counter which keeps track of number of pointerse for finding middle */

mid = start;
for (loop_counter=0; loop_counter < center; loop_counter++)
{
    if (mid == NULL) break;
    mid = mid->next;
}

/* Here mid is pointing to the middle elements */
return mid;
}

/* Binary Search on doubly linked list that returns a pointer to the position containing the key or
returns NULL */
DL_node * BinSearch(DL_node * start, DL_node * end , int query)
{
    DL_node * mid;

    if (start == NULL || end == NULL) return NULL;
    if ( !comes_after(start,end) ) return NULL;
    /* The above to return conditions for the recursion */

    mid = midpoint(start,end);
    if (mid == NULL) return NULL;
    /* Not a very meaningful condition, but just checking so that the
    next check is mid->value, and it must work meaningfully. A bit of
    DEFENSIVE programming */

    if (query == mid->value) return mid;
    /* query found, the position returned */

    /* recursion set-up */
    if (query > mid->value ) return BinSearch(mid->next,end,query);
    return BinSearch(start,mid->prev,query);
}
```

So, we have done with this part of the implementation, so we have augmented or added additional features to the doubly link list methods file, we added a binary search implementation assuming that data is in a doubly link list. Now, let us go to our comparison program, by the way I have also compile this, but I can compile this file gcc minus c dell istmethods dot c that compile and our dot o file has been generated. So, dot o file has been generated.

(Refer Slide Time: 32:43)

```
#include "DLList-interface.h"
#include <stdio.h>
#include <stdlib.h>

/* Developed by N.S. Narayanaswamy */
/* Function prototypes */
/* to check if the character corresponds to a digit */
DL_node * num_read(DL_node *, int *);
/* to read a number into an initial list */
void print_list(DL_node *);
int ABinSearch(DL_node *, int, int, int);
/* print the whole list */

int mid_probes=0;

int main()
{
    DL_node * number;
    int search_key;
    DL_node * number_array;
    int read_val;
    int array_size=0;
    int location;

    /* read number: */
    scanf("%d",&read_val);
    /* initial node of number */
    number = (DL_node *) malloc(sizeof(DL_node));
    number->value = read_val;
    number->next = NULL;
    number->prev = NULL;
    array_size++;
    number_array = num_read(number, &array_size);
    /* number_array and number contain the same number listing */

    scanf("%d",&search_key);

    "BinSchComp.c" 118L, 2998C
```

So, now let us go to our binary search comparison program, it should not take too long. I only have one additional function prototype here which is called a bin search in a doubly link list node array, array I repeat again int, int and int. The last int is the queried value and the first two ints are indices in this particular array and a stands for arrays. So, therefore I have a binary search implementation, assuming that the data is organized in an array and it returns a integer value which will be an index.

I also keep a global variable counter call mid probes which is going to count the total number of probes made into the array by the array binary search function. Everything is fairly straight forward let us just see this part. Here I am going to read the array, I am going to read the numbers, sequence into an array and the function is called num read, it is very interesting. So, what it does is it takes a pointer to the number doubly linked list that is the first argument.

The second argument is it takes an integer pointer pointed to the array size that is the first element, it is just a single number has been read. Array size is now 1, it takes a pointer to array size and it returns the number array and this number array is an array of doubly linked list nodes that contain exactly the same number sequence. I repeat this, the num read function is a very interesting function, it takes his argument a pointer to the doubly

linked list called number.

It also takes the pointer to an integer, this full long will contain the total number of values which have been read from the input that is what array size will consists of. It also returns a pointer to an array. What is the type of the array? The type of the array is a doubly linked list nodes. How many elements are there in this array? The number of doubly linked list nodes which are there in this array is exactly array size. What values are there? Exactly, the same values.

Why are we doing this? My goal remember is should compare binary search, when the data is showed in a doubly linked list or when the data is showed in an array. So, I am kindly taking a short cut to make my programming easy and I am storing the numbers in the number array, where the individual elements in each array are of type doubly linked dl node and number is in doubly linked list itself completely. We will go to the function in a short while, then I scan for the, then I read the search key from the input.

(Refer Slide Time: 35:50)

```
/* number_array and number contain the same number listing */
scanf("%d",&search_key);
print_list(number);

if (BinSearch(number,Dlast(number),search_key) != NULL) printf("Found\n");
printf("Cost of pointer traversals for mid probes = %d\n", finding_mid_binSearch);
print_list(number_array);

location = ABinSearch(number_array,0,array_size,search_key);
if (location > -1) printf("Found at %d \n", location);
printf("Number of mid probes in the array = %d\n",mid_probes);
return 0;
}

DL_node * num_read(DL_node * number, int * values_count)
{
    int read_val;
    DL_node * another_node;
    int index = 0;
    DL_node * ret_array;

    while (scanf("%d",&read_val))
    {
        (*values_count)++;
        another_node = (DL_node *) malloc(sizeof(DL_node));
        another_node->value = read_val;
        another_node->next = NULL;
        another_node->prev = Dlast(number);
        Dlast(number)->next = another_node;
    }

    /* This is very important, as the last data type read that failed to be converted to integer by
    scanf is sent back into the input stream and the stream must be flushed now */
    fpurge(stdin);
}
```

Now, I print the number list, then what do we do, I call binary search which is there in the doubly linked list methods dot c. I give it the number, I pointer to the first element in the doubly linked list and then I give a pointer to the last element in the doubly link list

for number, then I give the search key. If it is not equal to null then I print found. Remember that the binary search function returns a pointer to the location, if the key is found and when the key is not found and it returns the value null.

If it is not null it prints found, otherwise it just prints the cost of the total number of pointer traversals by printing out the value of finding mid binary search. Then, again it prints the contents of the number array that is the number sequence is present in this array. I perform an array bins a binary search on number array starting from the index zero up to array size which is the total number of keys, and such keys is another argument and this fellow returns the value into the variable called location.

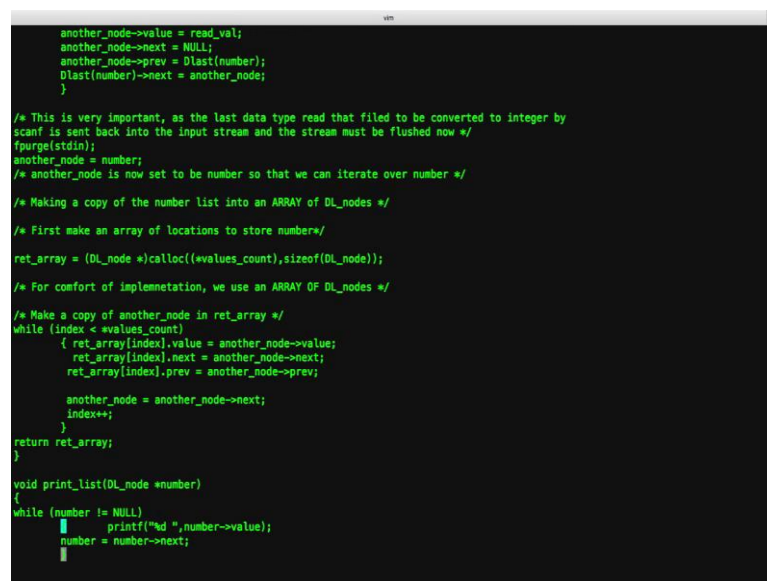
Indeed you will see that the implementation guarantees are at the key is not there, it returns a value minus 1. Otherwise, it returns the value in the range zero to array size minus 1. It print says it is found at a particular location and it prints the total number of mid probes which is the focus. Therefore, what are we done we have completed the implementation of binary search, one on an array the other on a doubly link list and we have printed out the statistics of the number of probes for finding the middle element and here the number of probes for the middle element which is the single arithmetic operation.

So, let us just quickly look at the read function, so let us just look at it. This is the pointer called values count, it is a pointer to the list of numbers, it is a doubly link list, list of numbers and the return value is a pointer to a doubly link list array which are called a return array, so let us just see this. You read an integer from the input buffer, so if the read was successful the read value will contain this that is it will have a positive value.

Remember that what scanf does. The return value of the scanf is the number of data items that it is successfully read. That is if you said read an integer, one integer then scanf will take the value one, if the read was successful. If the read fail that is if you put a non integer char, non numeric character there, then scanf will fail and this fellow will take the value zero. So, therefore while as long as scanf succeeded to read one integer that is what the meaning of the expression is, go ahead and do these steps. What are the steps?

You create a node of, you increase the size of the doubly link list. You create a node, you put the value inside the value field, then you make the next pointer go to null, then you make the previous pointer point to the last element of the current number list, then make the current number list point to make the newly added node the last element that is what you repeatedly do here.

(Refer Slide Time: 39:26)



```
another_node->value = read_val;
another_node->next = NULL;
another_node->prev = Dlast(number);
Dlast(number)->next = another_node;
}

/* This is very important, as the last data type read that failed to be converted to integer by
scanf is sent back into the input stream and the stream must be flushed now */
fpurge(stdin);
another_node = number;
/* another_node is now set to be number so that we can iterate over number */

/* Making a copy of the number list into an ARRAY of DL_nodes */
/* First make an array of locations to store numbers */
ret_array = (DL_node *)calloc((values_count),sizeof(DL_node));
/* For comfort of implementation, we use an ARRAY OF DL_nodes */
/* Make a copy of another_node in ret_array */
while (index < values_count)
{
    ret_array[index].value = another_node->value;
    ret_array[index].next = another_node->next;
    ret_array[index].prev = another_node->prev;

    another_node = another_node->next;
    index++;
}
return ret_array;
}

void print_list(DL_node *number)
{
    while (number != NULL)
    {
        printf("%d ",number->value);
        number = number->next;
    }
}
```

Now, here comes a bit of C programming which should be interesting for all of you. When control exits from this while loop, it means that this scanf failed. Scanf also what it does is if it fails, it returns the data the last value it sends it back into the input stream. Now, the input stream must be flushed, because after this we are going to read the search key, the way it is done is using this function call which is there in stdio call f purge.

If you programming on an unique system, then you should be able to get access to the f purge function call, please read it and you will understand what it does. So, it is because scanf sense the value back into the stream and we do not want that value in the stream, we want to throw it out, so we purge the std. Now, what do we do? Another node now points to number therefore, number my another node now point to the number list that we have read so far.

Now, what do we do? We know the total number of elements which are there in the input list, we create an array of that size, let us see this. This is the statement, the result is an array that is this is the pointer. Calloc is a contiguous alloc, it has two arguments. How many locations and what is the size of each location? The first argument is how many locations and the second argument is size of each location. Therefore, the result of calloc is that we have created an array of dl node data type.

And you may be worrying why I am doing this, it also shows you how to use calloc that is one purpose, but it is also the comfort of implementation. So, now let us see how to make a copy of the number list which we have read into this array that is what this loop does. I have initially index, initializes index variables to zero, I do this still index is less than or equal to the total number of values that we have return. All that I do is value is set to be another node dot value is it is essentially we iterate over the list and make a copy here.

So, let me just put a remark here, write in front of you, so that you appreciated, make a copy of another node in, this is what we have done, now you return, return. Now, let us look the result of execution or his function is very interesting. There are three side effects, three effects that are happen, you read the number into this doubly link list. You have also communicated the total number of values that you read by this particular pointer. Then, you also return the pointer to an array. There are three effects of this array and this is what is very interesting.

The kind of things that you can do, though people may believe that you can only return a single value from a pointer from a function, using pointers you can actually do fairly sophisticated dangerous things if you have not a careful program. So, this is the num read function as you can see it not only reads the whole list into an array, whole list into a doubly link list, but it also creates a copy of it in an array, it also returns the value of the number of keys that have been read. Print list is not any different from earlier, it just prints the contents of the doubly link list. In this case it can also print out the contents of the array that is the reason I have chosen the array of the same data type. I do not have to write another print function.

(Refer Slide Time: 43:45)

```
/* Make a copy of another_node in ret_array */
while (index < *values_count)
{
    ret_array[index].value = another_node->value;
    ret_array[index].next = another_node->next;
    ret_array[index].prev = another_node->prev;

    another_node = another_node->next;
    index++;
}
return ret_array;
}

void print_list(DL_node *number)
{
    while (number != NULL)
    {
        printf("%d ", number->value);
        number = number->next;
    }
    printf("\n");
    return;
}

/* Binary Search on an Array - returns the index of the locations where the
query is found */
int ABinSearch(DL_node *number_array, int start, int end, int query)
{
    int mid;
    if (start > end) return -1;
    mid_probes++;
    mid = (end + start)/2;
    /* A single arithmetic operation to calculate mid */
    /* Recall that in the Doubly linked List it was many pointer traversals */
    if (query == number_array[mid].value) return mid;
    if (query > number_array[mid].value) return ABinSearch(number_array, mid+1, end, query);
    return ABinSearch(number_array, start, mid-1, query);
}
```

Now, let us look at the binary search implementation, it is very straight forward, it is a same thing. Number array is passed to the function. Two specific indices, start and end are passed to the function and the query key. Now, have an integer variable called mid, so if start is more than end, I return minus 1, saying that the key is not there. This is indeed the correct exit condition for binary search in an array, otherwise start is less than or equal to n, I have to make one more probe to the mid of the list.

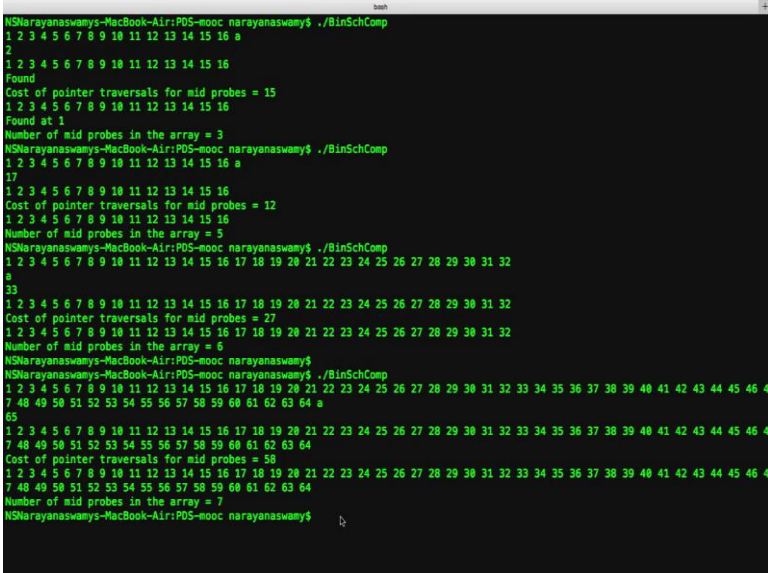
So, I increment the mid probe by 1, then I compute mid which is start plus end by 2, now I checked the queried value is in the mid location pointer to by mid. If it is there I return that particular integer value, otherwise the two recursive sub problems are generated exactly as we did in a doubly link list case, except that observe that mid is created by a single arithmetic operation to compute mid, I supposed to what it was, recall that in the doubly link list it was many pointer traversals.

Here it is this is a single arithmetic operation that is the only difference between the two implementations. So, that completes the whole program, let us see what we have done this program. We look at the doubly link list data type, we have added an additional binary search function tool, we are going to compare it with the array implementation therefore, we have a very interesting reading in function where we reading the same

sequence of sorted sequence of numbers into a doubly link list and into an array and then we implemented binary search on the array also.

So, now we are in a position to compare it and ((Refer Time: 45:59)) we already observe that we print out the number of probes and both the places. So, therefore we have ready to compile and execute this program, so the compilation is gcc BinSchCorp dot c, I want the output to go to BinSchCorp and I want to include, I want to list this particular object file which has been pre compile. So, compilation is succeeded now let us go to running this program dot BinSchCorp.

(Refer Slide Time: 46:39)



```
NSNarayanawams-MacBook-Air:PD5-mooc narayanawams$ ./BinSchComp
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 a
2
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
Found
Cost of pointer traversals for mid probes = 15
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
Found at 1
Number of mid probes in the array = 3
NSNarayanawams-MacBook-Air:PD5-mooc narayanawams$ ./BinSchComp
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 a
27
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
Cost of pointer traversals for mid probes = 12
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
Number of mid probes in the array = 5
NSNarayanawams-MacBook-Air:PD5-mooc narayanawams$ ./BinSchComp
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32
a
33
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32
Cost of pointer traversals for mid probes = 27
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32
Number of mid probes in the array = 6
NSNarayanawams-MacBook-Air:PD5-mooc narayanawams$ ./BinSchComp
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 4
7 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 a
65
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 4
7 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64
Cost of pointer traversals for mid probes = 58
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 4
7 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64
Number of mid probes in the array = 7
NSNarayanawams-MacBook-Air:PD5-mooc narayanawams$
```

So, it is waiting for an input, let us see what the input is let us give it in a sorted order 1, 2, 3, 4, 5, 6, 7, 8, 9, 10. Let me make it 11, 12, 13, 14, 15, 16 just the power of 2 so that you can see the effect of log. Now, the reading will fail when I give it a non integer, let me give it a. Now, a is mean read, the input array has been read, the input array has been read, now it is waiting for this search key, let us give the search key to be 2.

It is an easy check, observe that the responses that it has been found. The cost of pointer traversal it has made 15 pointer traversals, I will encourage you to do this by pen and paper to calculate that 15 pointer traversals are made. Then, the contents of num array

have been printed out, it says that it has been found at index one, but a number of mid probes is 3. Let us see, the midpoint of 16 size array is 8, then the midpoint of an 8 size array is 4, 2 for mid probes, an midpoint of 4 size array is 2 and the last probe will find the given key. This is for a successful search, let us do it for a failing search.

So, it is an important object for you to notice that the input has to be correctly given I am not checking that the input is in sorted format. If the input is not in sorted format this algorithm is not guarantee to work. Let us give the value 17 which is not present. So, as you can see that if you use an array with five probes, the array implementation discovers that the value is not in the list whereas, the total number of pointer traversals is almost the total number of elements in the list that is it is 12.

It is of the same order of the size of the list, whereas here it seems to be of the order of log of the size of the list. Let us do this for a larger array just to make this comparison, let us do this for 32 elements. So, you will see that let us do a failing search, let us search for the value 33, you have to print a non number which is a. Now, I have to print the value that I am searching for 33, as you can see it is made 27 pointer traversals, but the number of mid probes is only 6 that is one more than the case it was to 16.

Now, for 64 you can check it will only perform 7 and this should be almost of the order of 50. So, may be just for your benefit let me just type the whole thing down. 64 keys then the input, the whole list is been given let us say it as 65. As you can see the total number of probes is nearly the whole doubly link list is being probe that 58 probes, whereas observe that the array implementation as just makes 7 probes into the array to discover with the element is not found. So, this completes the whole discussion of the comparison of the binary search implementation on a doubly link list with the binary search implementation on an array.

(Refer Slide Time: 51:32)

3 data types, 2 algorithms

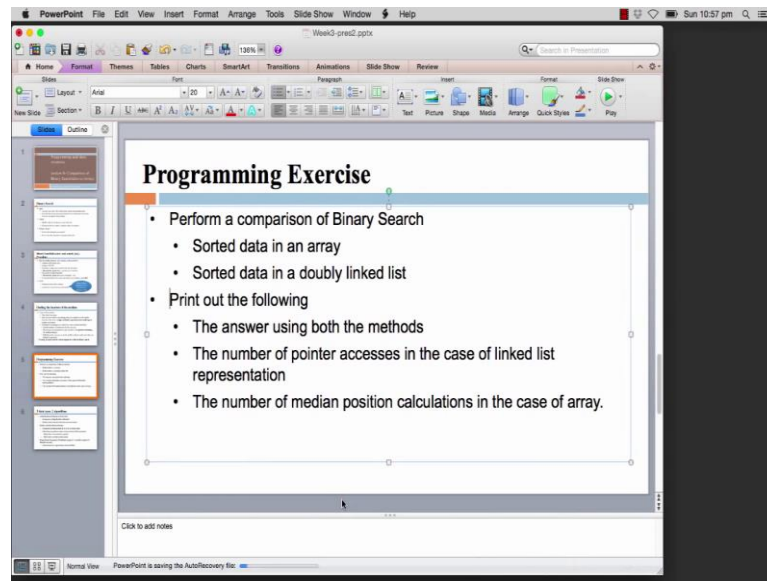
- Linked Lists and Doubly Linked Lists
 - **Compared on Big Number arithmetic**
 - Doubly Linked Lists are faster but use more space.
- Doubly Linked Lists and Arrays
 - **Compared on Binary Search of a list of sorted data**
 - With Arrays we perform lesser number of basic CPU operations
 - Middle index by one arithmetic operation
 - Middle index by multiple pointer queries.
 - Arrays have the power of hardware support + compiler support of Random Access.
 - Access based on a given base value and offset

So, it is quite clear that the binary search implementation on an array is very efficient in terms of the total number of probes. So, therefore what are we done during this week, we have study three data types link list, doubly link list and arrays. We have taking two algorithms. One algorithm was the big number arithmetic we did only big number addition. But you will recall that the polynomial multiplication exercise that I given is also very simple more general big number arithmetic.

Link list and doubly link list are combined, we notice that the doubly linked lists are faster, but use more space. Then, we compare doubly link list and arrays for binary search of a list of sorted data. With arrays we perform lesser number of basic CPU operations that is we find the middle index based on one arithmetic operation whereas, in a doubly link list we find the middle index by multiple pointer queries.

Therefore, one can imagine that arrays have the power of hardware support and the compiler support of random access and the access based on a given base value and an offset. This is not very important, but if you understand the bit of compute organization if we have already studied it, you will immediately appreciate.

(Refer Slide Time: 52:57)



So, therefore when we meet in the next set of lectures, we are going to look at the abstract data types, stack and queue. And we will see how recursive algorithms can be implemented without the use of recursion, but with the use of a stack. So, that is going to be the focus for the next week and this brings to an end today's lecture.

Thank you very much.