Data Structures & Algorithms

Data Structures

- Text Book
  - C++ Introduction to Data Structures by Larry Nayhoff
  - Data Structures and Algorithm Analysis in C++
    Mark Allen Weiss, *Florida International University*
    Addison-Wesley
- Reference Material
  - Data Structures A psuedocode Approach with C by Richard F.Gilberg & Behrouz A.Forouzan
  - Object oriented programming in C++ by Robert Lafore

Data Structure and Algorithm

- Data structures offer different ways to store data items.
- While the algorithms provide techniques for managing data. For example, there are many algorithms to sort data.
- Without one, the other is useless. Together, they make computer programs. They're both fundamental.

Need for Data Structures

- Data structures organize data $\Rightarrow$ more efficient programs.
- More powerful computers $\Rightarrow$ more complex applications.
- More complex applications demand more calculations.

Organizing Data

- Any organization for a collection of records that can be searched, processed in any order, or modified.
- The choice of data structure and algorithm can make the difference between a program running in a few seconds or many days.

Efficiency

- A solution is said to be efficient if it solves the problem within its resource constraints.
  - Space
  - Time

- The cost of a solution is the amount of resources that the solution consumes.

Selecting a Data Structure

Select a data structure as follows:

1. Analyze the problem to determine the resource constraints a solution must meet.
2. Determine the basic operations that must be supported. Quantify the resource constraints for each operation.
3. Select the data structure that best meets these requirements.

Some Questions to Ask

- Are all data inserted into the data structure at the beginning, or are insertions scattered with other operations?
- Can data be deleted?
- Are all data processed in some well-defined order, or is random access allowed?

Data Structure Philosophy

- Each data structure has costs and benefits.
- Rarely is one data structure better than another in all situations.
- A data structure requires:
  - space for each data item it stores,
  - time to perform each basic operation,
  - programming effort.

Arrays

- Elementary data structure that exists as built-in in most programming languages.

```
main( int argc, char** argv )
{
    int x[6];
    int j;
    for(j=0; j < 6; j++)
            x[j] = 2*j;
}
```

Arrays

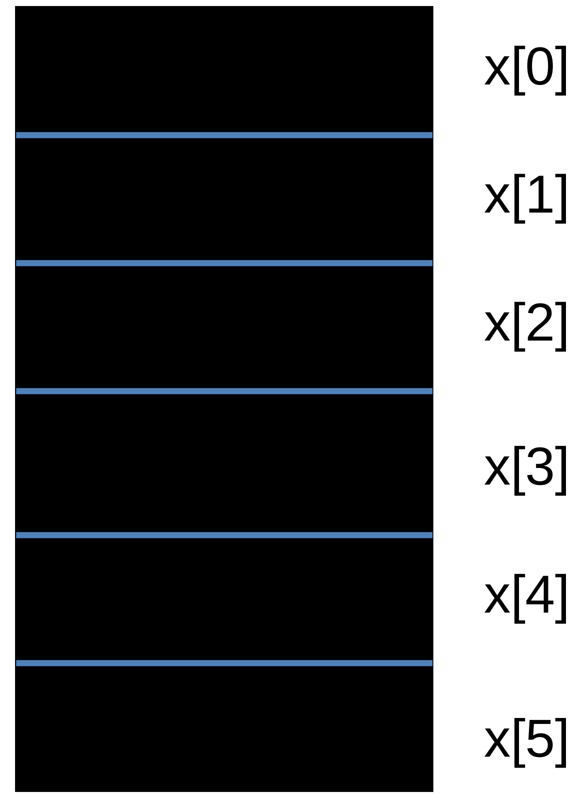
- Array declaration: int x[6];
- An array is collection of cells of the same type.
- The collection has the name ‘x’.
- The cells are numbered with consecutive integers.
- To access a cell, use the array name and an index:

  x[0], x[1], x[2], x[3], x[4], x[5]

Array Layout

Array cells are contiguous in computer memory

The memory can be thought of as an array



What is Array Name?

- ‘x’ is an array name but there is no variable x. ‘x’ is not an lvalue.
- For example, if we have the code

```
int a, b;
```

then we can write

```
b = 2;
a = b;
a = 5;
```

But we cannot write

```
2 = a;
```

Array Name

- ‘x’ is not an lvalue

```
int x[6];
int n;

x[0] = 5;
x[1] = 2;

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

Dynamic Arrays

- You would like to use an array data structure but you do not know the size of the array at compile time.
- You find out when the program executes that you need an integer array of size n=20.
- Allocate an array using the new operator:

```
int* y = new int[20];  // or int* y = new int[n]
y[0] = 10;
y[1] = 15;                 // use is the same
```

Dynamic Arrays

- ‘y’ is a lvalue; it is a pointer that holds the address of 20 consecutive cells in memory.
- It can be assigned a value. The new operator returns as address that is stored in y.
- We can write:

```
y = &x[0];
y = x;        // x can appear on the right
              // y gets the address of the
              // first cell of the x array
```

Dynamic Arrays

- We must free the memory we got using the new operator once we are done with the *y* array.

```
delete[ ] y;
```

- We would not do this to the x array because we did not use new to create it.

The LIST Data Structure

- The List is among the most generic of data structures.

- Real life:

    a. shopping list,
    b. groceries list,
    c. list of people to invite to dinner
    d. List of presents to get

Lists

- A list is collection of items that are all of the same type (grocery items, integers, names)
- The items, or elements of the list, are stored in some particular order
- It is possible to insert new elements into various positions in the list and remove any element of the list

Lists

- List is a set of elements in a linear order.
  For example, data values a_1, a_2, a_3, a_4 can be arranged in a list:

  (a_3, a_1, a_2, a_4)

  In this list, a_3, is the first element, a_1 is the second element, and so on

- The order is important here; this is not just a random collection of elements, it is an ordered collection

List Operations

Useful operations

- createList(): create a new list (presumably empty)
- copy(): set one list to be a copy of another
- clear(); clear a list (remove all elments)
- 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(): return the length of the list.

List Operations

- We need to decide what is meant by “particular position”; we have used “?” for this.

- There are two possibilities:

1. Use the actual index of element: insert after element 3, get element number 6. This approach is taken by arrays
2. Use a “current_Index” marker or pointer to refer to a particular position in the list.

List Operations

- If we use the “current_Index_Index” marker, the following four methods would be useful:

  - start(): moves to “current_Index_Index” pointer to the very first element.
  - tail(): moves to “current_Index” pointer to the very last element.
  - next(): move the current_Index position forward one element
  - back(): move the current_Index position backward one element

Implementing Lists

- We have designed the interface for the List; we now must consider how to implement that interface.
- Implementing Lists using an array: for example, the list of integers (2, 6, 8, 7, 1) could be represented as:

A	2	6	8	7	1			
	1	2	3	4	5			

current_Index	size
3	5

List Implementation

- add(9); current_Index position is 3. The new list would thus be: (2, 6, 8, 9, 7, 1)
- We will need to shift everything to the right of 8 one place to the right to make place for the new element ‘9’.

step 1:

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

current_Index	size
3	5

step 2:

A	2	6	8	**9**	7	1		
	1	2	3	4	5	6		

current_Index	size
4	6

notice: current_Index points to new element

Implementing Lists

- next():

A	2	6	8	**9**	7	1		
	1	2	3	4	5	6		5

current_Index	size
4	6

Implementing Lists

- next():

A	2	6	8	**9**	7	1		
	1	2	3	4	5	6		

current_Index: ~~4~~ 5

size: 6

- There are special cases for positioning the current_Index pointer:
  a. past the last array cell
  b. before the first cell

- We will have to worry about these when we write the actual code.

Implementing Lists

- remove(): removes the element at the current_Index

Step 1:

index

A

2	6	8	9		1		
1	2	3	4	5	6		

current_Index	size
5	~~6~~ 5

Step 2:

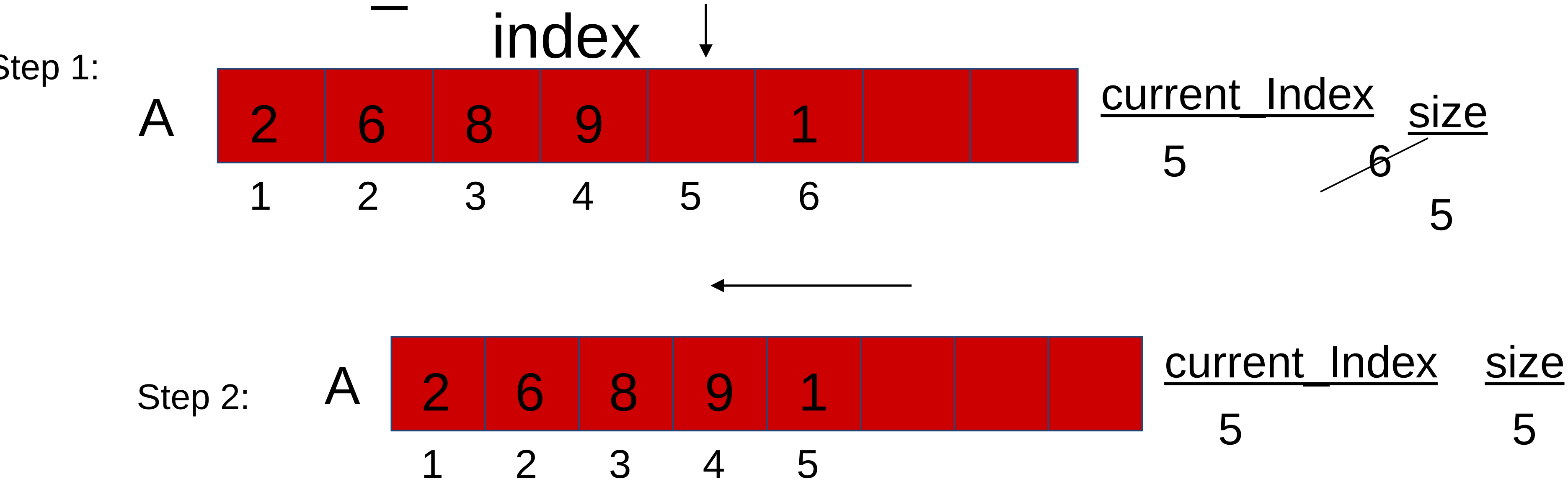
A

2	6	8	9	1			
1	2	3	4	5			

current_Index	size
5	5

Implementing Lists

- remove(): removes the element at the current_Index



- We fill the blank spot left by the removal of 7 by shifting the values to the right of position 5 over to the left one space.

Implementing Lists

find(X): traverse the array until X is located.

```
int find(int X)
{
    int j;
    for(j=1; j < size+1; j++ )
        if( A[j] == X ) break;

    if( j < size+1 ) {        // found X
        current_Index = j;        // current_Index points to
where X found
        return 1;   // 1 for true
    }
    return 0;  // 0 (false) indicates not found
}
```

Implementing Lists

- Other operations:

get() → return A[current_Index];
update(X) → A[current_Index] = X;
length() → return size;
start() → current_Index=0;
end() → current_Index=size;

Analysis of Array Lists

- insert
  - we have to move every element to the right of provided location to make space for the new element.
  - Worst-case is when we insert at the beginning; we have to move every element right one place.
  - Average-case: on average we may have to move half of the elements

Analysis of Array Lists

- remove
  - Worst-case: remove at the beginning, must shift all remaining elements to the left.
  - Average-case: expect to move half of the elements.
- find
  - Worst-case: may have to search the entire array
  - Average-case: search at most half the array.
- Other operations are one-step.