Data Structure and Algorithms Lecture 1

Lecture 1

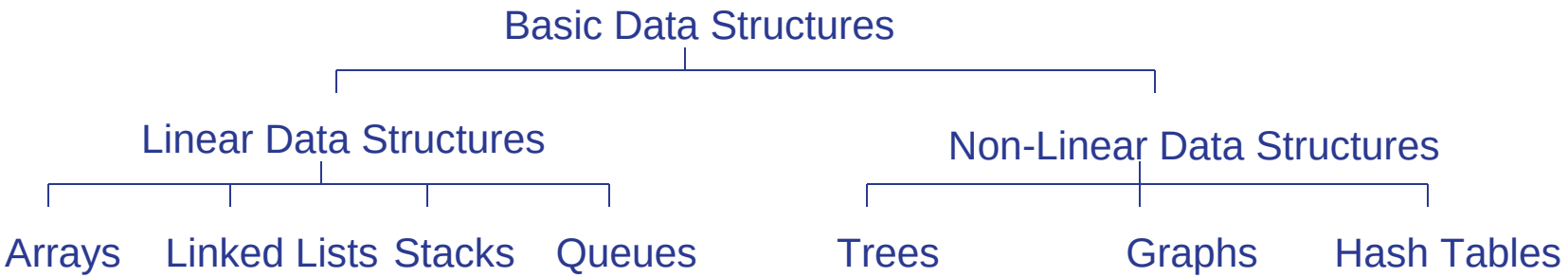
What is Data Structure?

- Data structure is a representation of data and the operations allowed on that data.

- A data structure is a way to store and organize data in order to facilitate the access and modifications.

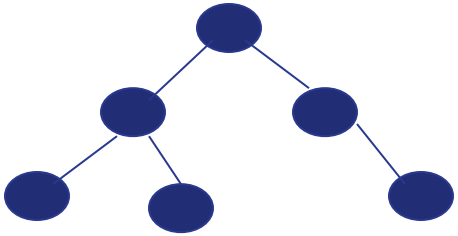
- Data Structure are the method of representing of logical relationships between individual data elements related to the solution of a given problem.

Basic Data Structure



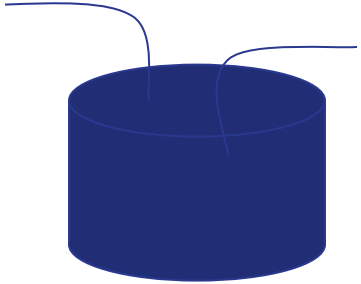
array

Linked list



tree

queue



stack

Types of Data Structure

- Linear: In Linear data structure, values are arrange in linear fashion.
  - Array: Fixed-size
  - Linked-list: Variable-size
  - Stack: Add to top and remove from top
  - Queue: Add to back and remove from front
  - Priority queue: Add anywhere, remove the highest priority

Types of Data Structure

- Non-Linear: The data values in this structure are not arranged in order.
  - Hash tables: Unordered lists which use a ‘hash function’ to insert and search
  - Tree: Data is organized in branches.
  - Graph: A more general branching structure, with less strict connection conditions than for a tree

Type of Data Structures

- Homogenous: In this type of data structures, values of the same types of data are stored.
  - Array

- Non-Homogenous: In this type of data structures, data values of different types are grouped and stored.
  - Structures
  - Classes

Structure

- Definition:-
  - *Abstract Data Types (ADTs)* stores data and allow various operations on the data to access and change it.
  - A mathematical model, together with various operations defined on the model
  - An ADT is a collection of data and associated operations for manipulating that data

- Data Structures
  - Physical implementation of an ADT
  - data structures used in implementations are provided in a language *(primitive* or *built-in)* or are built from the language constructs *(user-defined)*

Selection of Data Structure

- The choice of particular data model depends on two consideration:
  - It must be rich enough in structure to represent the relationship between data elements
  - The structure should be simple enough that one can effectively process the data when necessary

Abstract Data Type

- ADTs support *abstraction*, *encapsulation*, and *information hiding*.

- *Abstraction* is the structuring of a problem into well-defined entities by defining their data and operations.

- The principle of hiding the used data structure and to only provide a well-defined interface is known as *encapsulation*.

The Core Operations of ADT

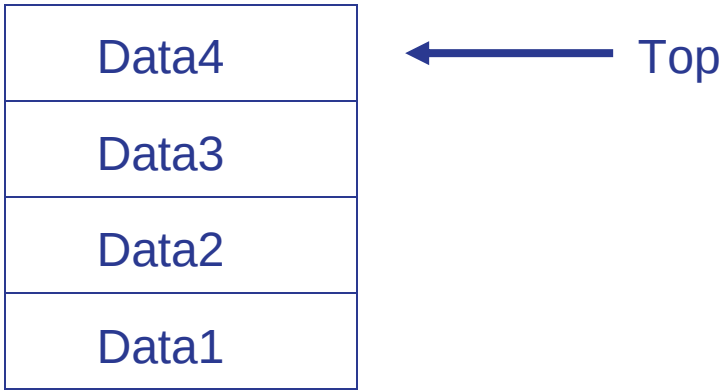
- Every Collection ADT should provide a way to:
  - add an item
  - remove an item
  - find, retrieve, or access an item

- Many, many more possibilities
  - is the collection empty
  - make the collection empty

- No single data structure works well for all purposes, and so it is important to know the strengths and limitations of several of them

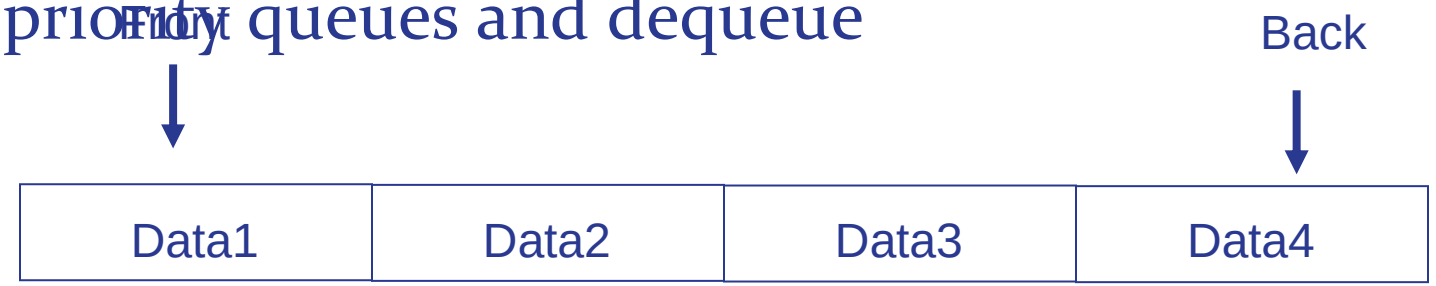
Stacks

- Collection with access only to the last element inserted
- Last in first out
- insert/push
- remove/pop
- top
- make empty



Queues

- Collection with access only to the item that has been present the longest
- Last in last out or first in first out
- enqueue, dequeue, front
- priority queues and dequeue



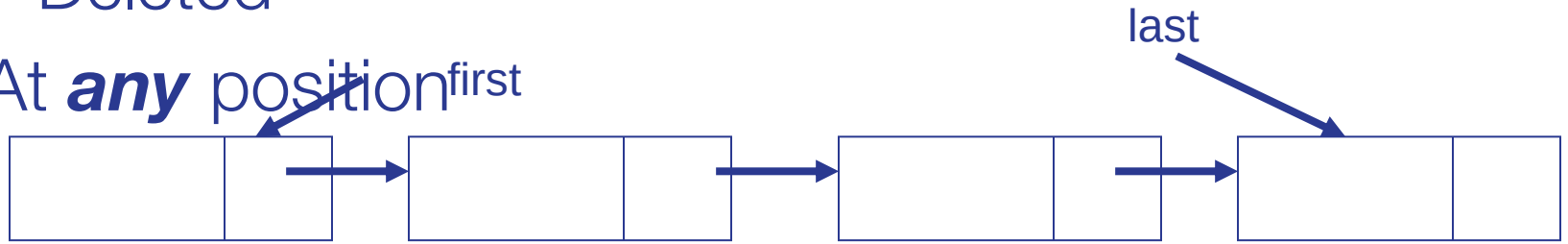
List

- A ***Flexible*** structure, because can grow and shrink on demand.

Elements can be:

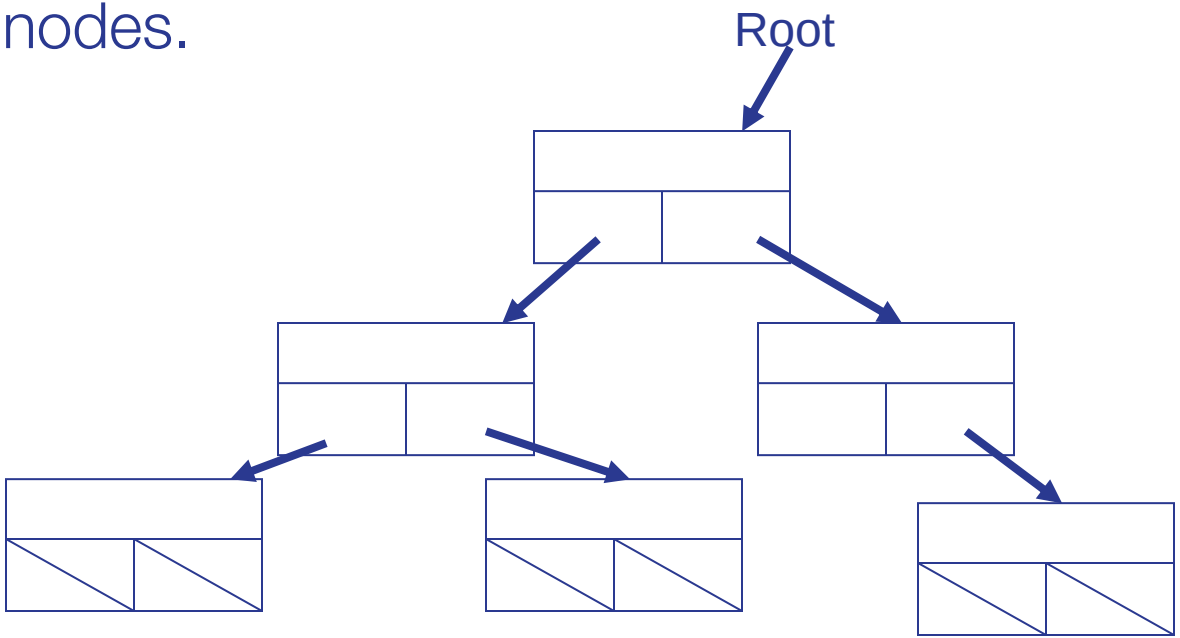
- Inserted
- Accessed
- Deleted

At ***any*** position



Tree

- A ***Tree*** is a collection of elements called ***nodes.***
- One of the node is distinguished as a ***root***, along with a relation (“parenthood”) that places a hierarchical structure on the nodes.



Structured Data Type - Array

- Data types examined so far are atomic:
  - int, long
  - float, double
  - char
- Called “simple” data types because vars hold a single value
- If limited to “simple” data types, many programming applications are tedious
- Solution: use structured data types – types

Outline

Structured Types

A. Arrays

1. Syntax of declaration
2. Layout in memory
3. Referencing array element
    a. Subscript use (abuse)
    b. Array elements as variables
4. Array Initialization

Outline (cont)

- Structured Types
  - A. Arrays (cont)
    - 6. Passing array/part of array
      - a. element of array
      - b. entire array
    - 7. Multidimensional arrays
      - a. declaration
      - b. referencing
      - c. processing

Outline (cont)

Techniques

- A. Sorting
  1. What is sorted?
  2. Selection sort
  3. Insertion sort
- B. Searching
  1. (Un)successful searches
  2. Sequential search
     a. Unsorted array

Sample Problem

Problem: track sales totals for 10 people

Daily data:

Employee #	Sale
3	9.99
7	16.29
9	7.99
3	2.59
.	
.	
.	
7	49.00

Representing Sales Data

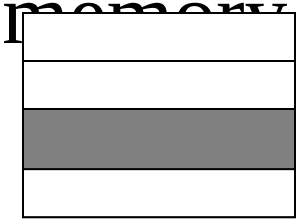
```
FILE* sdata;
int numE;
float amtS;
float S0, S1, S2, S3, S4, S5, S6, S7, S8,
  S9 = 0.0; /* One var for each employee */

if ((sdata = fopen(“DailySales”,”r”)) ==
  NULL) {
  printf(“Unable to open DailySales\n”);
  exit(-1);
```

Updating Sales Data

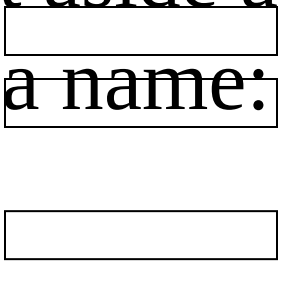
```
while (fscanf(sdata,”%d%f”,&numE,&amtS)
  == 2) {
  switch (numE) {
    case 0: S0 += amtS; break;
    case 1: S1 += amtS; break;
    case 2: S2 += amtS; break;
    case 3: S3 += amtS; break;
    case 4: S4 += amtS; break;
    case 5: S5 += amtS; break;
    case 6: S6 += amtS; break;
    case 7: S7 += amtS; break;
    case 8: S8 += amtS; break;
    case 9: S9 += amtS; break;
```

Data Structure

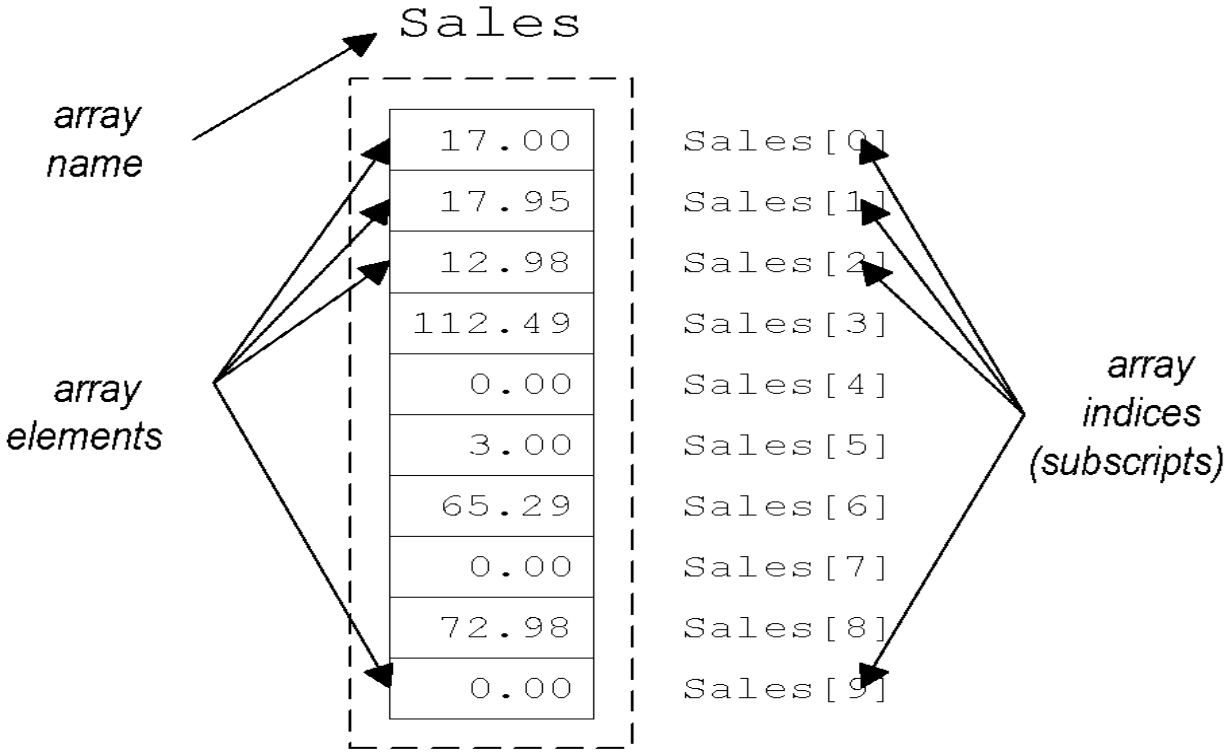
- A *Data Structure* is a *grouping* of data items in memory under one name
- When data items same type, can use *Array*
- Using an array, we can set aside a block of memory giving the block a name:



vs.



Parts of an Array



Declaring a 1D Array

Syntax: *Type Name*[*IntegerLiteral*]

- Type can be any type we have used so far
- Name is a variable name used for the whole array
- Integer literal in between the square brackets ([]) gives the size of the array (number of sub-parts)
- Size must be a constant value (no variable size)
- Parts of the array are numbered starting from 0
- 1-Dimensional (1D) because it has one index

Example:

- float Sales[10];

Array Indices

- The array indices are similar to the subscripts used in matrix notation:
  `Sales[0]` is C notation for $Sales^{0}$
  `Sales[1]` is C notation for $Sales^{1}$
  `Sales[2]` is C notation for $Sales_{2}$
- The index is used to refer to a part of array
- Note, C does not check your index (leading

Accessing Array Elements

- Requires
  - array name,
  - subscript labeling individual element
- Syntax: *name*[*subscript*]
- Example

  Sales[5] refers to the sales totals for employee 5

  Sales[5] can be treated like any float variable:

  Sales[5] = 123.45;

Invalid Array Usage

Example:

```
float Sales[10];
```

Invalid array assignments:

```
Sales = 17.50;          /* missing subscript */
Sales[-1] = 17.50;      /* subscript out of range */
Sales[10] = 17.50;      /* subscript out of range */
Sales[7.0] = 17.50;     /* subscript wrong type */
Sales[‘a’] = 17.50;     /* subscript wrong type */
Sales[7] = ‘A’;         /* data is wrong type */
```

Conversion will still occur:

Array Initialization

- Arrays may be initialized, but we need to give a *list* of values
- Syntax:

  Type Name[*Size*] =

  { *value0*, *value1*, *value2*, …, *valueSize-1* };

  value0 initializes *Name*[0], *value1* initializes *Name*[1], etc.

Array Initialization

Example:

```
int NumDays[12] = { 31, 28, 31, 30, 31, 30, 31,
   31, 30, 31, 30, 31 };  /* Jan is 0, Feb is 1, etc. */
int NumDays[13] = { 0, 31, 28, 31, 30, 31, 30,
   31, 31, 30, 31, 30, 31 }; /* Jan is 1, Feb is 2, */
```

Note:

if too few values provided, remaining array members not initialized

Sales Data as an Array

Recall data:

Employee #	Sale
3	9.99
7	16.29
9	7.99
3	2.59
.	
.	
.	
7	49.00

Idea: declare Sales as array, update array items

Updating Sales Data

```
while (fscanf(sdata,”%d%f”,&numE,&amtS)
  == 2) {
   switch (numE) {
     case 0: Sales[0] += amtS; break;
     case 1: Sales[1] += amtS; break;
     case 2: Sales[2] += amtS; break;
     /* cases 3-8 */
     case 9: Sales[9] += amtS; break;
   }
}
```

Q: What’s the big deal??

A: Can replace entire switch statement with:

Updating with Arrays

```
while (fscanf(sdata,”%d%f”,&numE,&amtS)
  == 2) {
  Sales[numE] += amtS;
}
```

When referencing array element can use any expression producing integer as subscript

[] is an operator, evaluates subscript expression then the appropriate location from the array is found

Note, when we have an integer expression, we may want to check subscript before using:

```
if ((numE >= 0) && (numE <= 9))
  Sales[numE] += amtS;
```

Using Array Elements

Array elements can be used like any variable

read into:

```
printf("Sales for employee 3: ");
scanf("%f",&(Sales[3]));
```

printed:

```
printf("Sales for employee 3 $%7.2f\n",Sales[3]);
```

used in other expressions:

```
Total = Sales[0] + Sales[1] + Sales[2] +
```

Arrays and Loops

- Problem: initialize Sales with zeros

  Sales[0] = 0.0;

  Sales[1] = 0.0;

  …

  Sales[9] = 0.0;

- Should be done with a loop:

  for (I = 0; I < 10; I++)

Processing All Elements of Array

- Process all elements of array A using for:

  /* Setup steps */

  for (I = 0; I < *ArraySize*; I++)

    process A[I]

  /* Clean up steps */

- Notes

  I initialized to 0

Initialize with Data from File

```
if ((istream =
   fopen(“TotalSales”,”r”))
   == NULL) {
  printf(“Unable to open
   TotalSales\n”);
  exit(-1);
}

for (I = 0; I < 10; I++)
  fscanf(istream,”%f”,
  &(Sales[I]));
```

- File TotalSales

  1276.17 (Emp 0’s Sales)

  917.50 (Emp 1’s Sales)

  …

  2745.91 (Emp 9’s Sales)

Array Programming Style

- Define constant for highest array subscript:

  #define MAXEMPS 10
- Use constant in array declaration:

  float Sales[MAXEMPS];
- Use constant in loops:

  for (I = 0; I < MAXEMPS; I++)

    fscanf(istream,”%f”,&(Sales[I]));
- If MAXEMPS changes, only need to

Summing Elements in an Array

```
Sales
117.00  Sales[0]
129.95  Sales[1]
276.22  Sales[2]
…
197.81  Sales[9]
```

```
I     Total
       0.00
0    117.00 (Emp 0 sales)
1    246.95 (0 + 1 sales)
2    523.17 (0 + 1 + 2 s)
…
9   1943.89 (All emps)
```

```
total = 0.0;
for (I = 0; I < MAXEMPS; I++)
  total += Sales[I];
```

Maximum Element of an Array

```
Sales
117.00  Sales[0]
129.95  Sales[1]
276.22  Sales[2]
…
197.81  Sales[9]
```

```
I      maxS
       117.00
1      129.95 (Max of 0,1)
2      276.22 (Max of 0,1,2)
…
9      276.22 (Max of all)
```

```
maxS = Sales[0];
for (I = 1; I < MAXEMPS; I++)
  if (Sales[I] > maxS)
    maxS = Sales[I];
```

Printing Elements of an Array

```
Sales
117.00  Sales[0]
129.95  Sales[1]
276.22  Sales[2]
…
197.81  Sales[9]
```

```
printf("Emp Num     Sales\n");
printf("-------     -----\n");
for (I = 0; I < MAXEMPS; I++)
  printf("%4d%13.2f\n", I,
```

Output:

Emp Num	Sales
0	117.00
1	129.95
2	276.22
…	
9	197.81

Passing an Element of an Array

- Each element of array may be passed as parameter as if it were variable of base type of array (type array is made of)
- When passing array element as reference parameter put & in front of array element reference ([] has higher precedence)
  - does not hurt to put parentheses around array

Passing Array Element Examples

Passing by value:

```
void printEmp(int eNum, float
   eSales) {
  printf("%4d%13.2f\n", eNum,
   eSales);
}

in main:

printf("Emp Num     Sales\n");
printf("-------     -----\n");
for (I = 0; I < MAXEMPS; I++)
```

Passing by reference:

```
void updateSales(float
   *eSales, float newS) {
  *eSales = *eSales + newS;
}

in main:

while
   (fscanf(sdata,"%d%f",
   &numE,&amtS) == 2)
   updateSales(
```

Passing Whole Array

- When we pass an entire array we always pass the address of the array
  - syntax of parameter in function header:

    BaseType NameforArray[]
    - note, no value between [] - compiler figures out size from argument in function call
  - in function call we simply give the name of the array to be passed
  - since address of array is passed, changes to

Passing Whole Array Example

```
float calcSalesTotal(float S[]) {
  int I;
  float total = 0.0;

  for (I = 0; I < MAXEMPS; I++)
    total += S[I];

  return total;
}

in main:
```

Size of an Array

- Sometimes we know we need an array, but not how big the array is going to be
- One solution:
  - allocate a very large array
  - integer to indicate num elements of array in use
  - loops use num elements when processing
- Example:

```
float Sales[MAXEMPS];
int NumEmps = 0;
```

processing:

Sorting Arrays

- One common task is to sort data
- Examples:
  - Phone books (by name)
  - Checking account statement (by check #)
  - Dictionaries (by word)
  - NBA scoring leaders (by points)
  - NBA team defense (by points)

Sorting Order

Team defense

90.0	Bulls
90.3	Heat
91.0	Knicks
…	
98.9	76ers

Sorted in *ascending* order

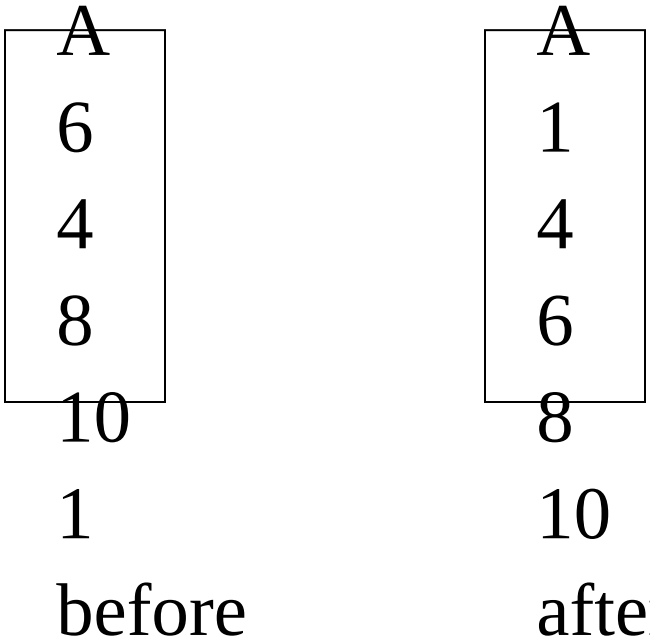
Individual Scoring

31.0	Jordan
28.7	Malone
27.9	O’Neill
…	
0.3	Bricklayer

Sorted in *descending* order

Sorting in Arrays

Data sorted usually same type, sorted in array

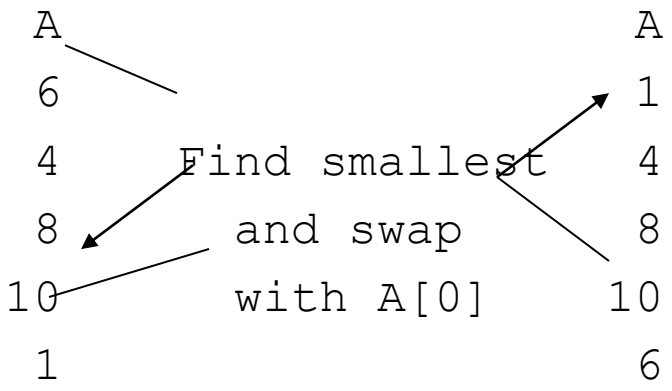


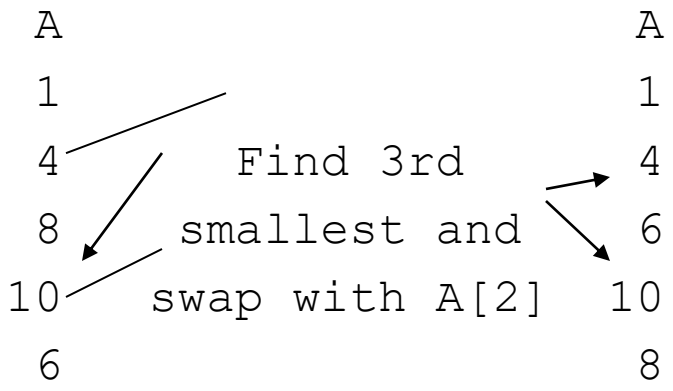
Selection Sorting

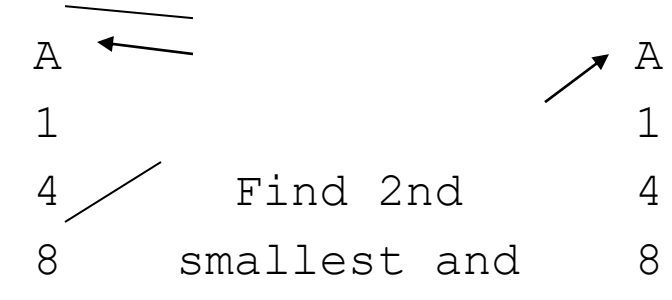
One approach (N is # elements in array):

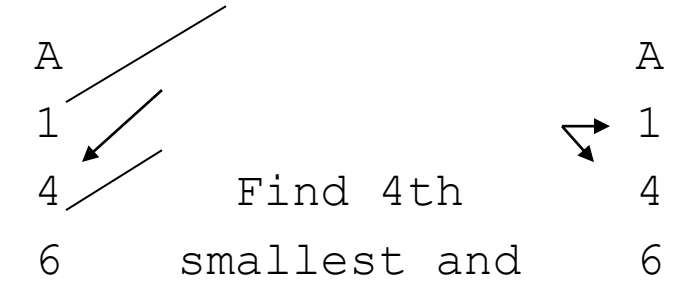
1. Find smallest value in A and swap with A[0]
2. Find smallest value in A[1] .. A[N-1] and swap with A[1]
3. Find smallest value in A[2] .. A[N-1] and swap with A[2]
4. Continue through A[N-2]

Selection Sort Example









Selection Sort Notes

- If array has N elements, process of finding smallest repeated N-1 times (outer loop)
- Each iteration requires search for smallest value (inner loop)
- After inner loop, two array members must be swapped
- Can search for largest member to get

Selection Sort Algorithm

For J is 0 to N-2

- Find smallest value in A[J], A[J+1] .. A[N-1]
- Store subscript of smallest in Index
- Swap A[J] and A[Index]

Find smallest in A[J..N-1]

- Suppose J is 0
  - Smallest = A[0];
  - for (K = 1; K < N; K++)

A	K	Smallest
6		6
4	1	4
8	2	
10	3	
1	4	1

But we need location of

Selection Sort Algorithm

Find location of smallest rather than value:

```
SmallAt = 0;
for (K = 1; K < N; K++)
   if (A[K] < A[SmallAt])
      SmallAt = K;
```

Swapping two elements:

```
Temp = A[J];
A[J] = A[SmallAt];
```

J is 0, find smallest:

A	K	SmallAt
6		0
4	1	1
8	2	
10	3	
1	4	4

Swap A[SmallAt],A[0]

Code for Selection Sort

```
for (J = 0; J < N-1; J++) { /* 1 */
  SmallAt = J;              /* 2 */
  for (K = J+1; K < N; K++) /* 3 */
    if (A[K] < A[SmallAt])  /* 4 */
      SmallAt = K;          /* 5 */
  Temp = A[J];              /* 6 */
  A[J] = A[SmallAt];        /* 7 */
```

Selection Sort Example

S#	J	K	Sml	Effect
1	0			Start outer
2			0	Init SmallAt
3		1		Start inner
4				True, do 5
5			1	Update SmallAt
3		2		Repeat inner
4				False, skip 5
3		3		Repeat inner
4				False, skip 5
3		4		Repeat inner
4				True, do 5
5			4	Update SmallAt
6-8				Swap A[0],A[4]
1	1			Repeat outer
2			1	Init SmallAt
4				False, skip 5
6-8				Swap A[1],A[1]
1	2			Repeat outer
2			2	Init SmallAt
3		3		Start inner
4				False, skip 5
3		4		Repeat inner
4				True, do 5
5			4	Update SmallAt
6-8				Swap A[2],A[4]
1	3			Repeat outer
2			3	Init SmallAt
3		4		Start inner

Modularizing Sort

- Want to make sort more readable
- Also, make sort a separate function
- First make Swap a separate function
- Have to pass array elements as reference

```
void Swap(int *n1, int *n2) {
  int temp;

  temp = *n1;
  *n1 = *n2;
  *n2 = temp;
}

for (J = 0; J < N-1; J++) {
  SmallAt = J;
  for (K = J+1; K < N; K++)
    if (A[K] < A[SmallAt])
      SmallAt = K;
```

Modularizing Sort - Find Smallest

- Can make process of finding smallest a separate function
- Sort becomes:

```
for (J = 0; J < N-1; J++)
{
  SmallAt =
    findSmallest(A,J,N);
  Swap(&(A[J]),
       &(A[SmallAt]));
```

```
int findSmallest(
  int Aname[], int J, int N)
{
  int MinI = J;
  int K;

  for (K = J+1; K < N; K++)
    if (Aname[K] <
        Aname[MinI])
      MinI = K;

  return MinI;
}
```

Modularizing Sort Itself

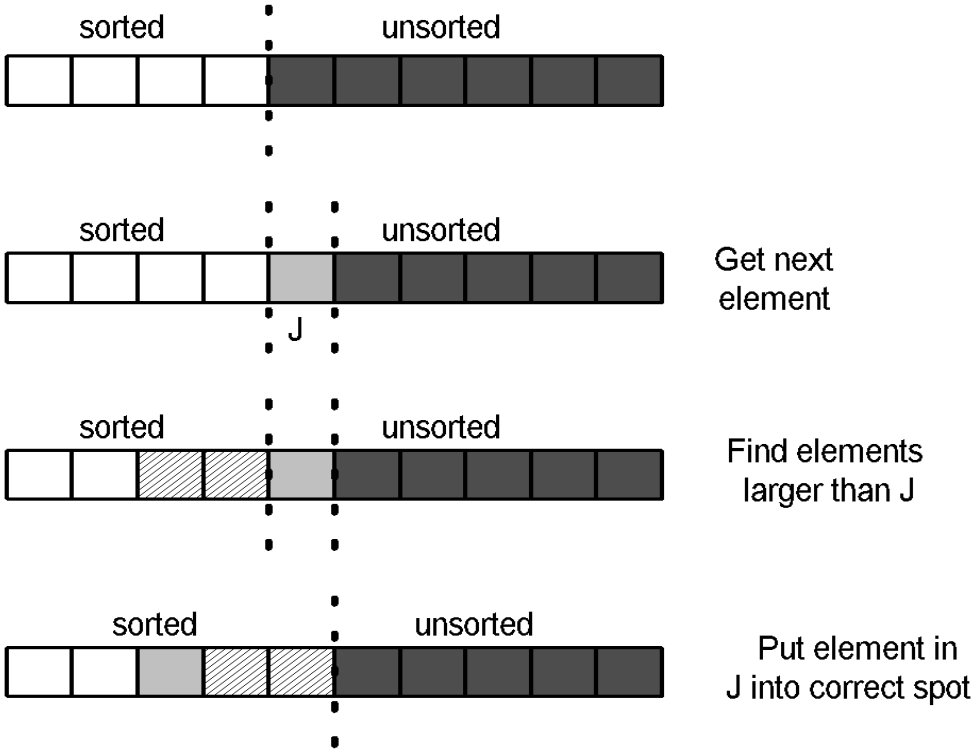
- Can then make the sort routine itself a function
- Localize variables such as J, SmallAt in function
- Pass N only if different from size of

```
void sort(int A[], int N) {
  int J;
  int SmallAt;

  for (J = 0; J < N-1; J++)
  {
    SmallAt =
      findSmallest(A,J,N);
    Swap(&(A[J]),
         &(A[SmallAt]));
  }
}
```

Another Sort: Insertion

- Idea: sort like a hand of cards
- Algorithm:
  - Assume A[0] .. A[0] is sorted segment
  - Insert A[1] into sorted segment A[0 .. 0]
  - Insert A[2] into sorted segment A[0 .. 1]
  - Insert A[3] into sorted segment A[0 .. 2]
  - continue until A[N-1] inserted

Inserting into Sorted Segment



Insertion Sort Code

```
void insertIntoSegment(int Aname[], int J) {
  int K;
  int temp = Aname[J];

  for (K = J; (K > 0) && (Aname[K-1] > temp); K--)
    Aname[K] = Aname[K-1];

  Aname[K] = temp;
}

void sort(int A[], int N) {
  int J;

  for (J = 1; J < N; J++)
```

Searching Arrays

Prob: Search array A (size N) for value Num

Example:

search array for a particular student score

```
A       N
6       5
4
8       Num
10      8
1
```

Sequential search: start at beginning of array,

Sequential Search

- Uses:
  - event-controlled loop
  - must not search past end of array
- Code:

```
index = 0;                                    /* 1 */
while ((index < N) && (A[index] != Num)) /* 2 */
  index++;                                    /* 3 */
```

- When loop finishes either:
  - index is location of value or
  - index is N
- Test

```
if (index < N)                                /* 4 */
```

Sequential Search Example

A	N	S#	Num	N	index	effect
6	5		8	5		
4		1			0	init index
8	Num	2				true, do 3
10	8	3			1	inc index
1		2				true, do 3
		3			2	inc index
		2				false, exit
		4				true, do 5
		5				print

Sequential Search Example

A	N
6	5
4	
8	Num
10	5
1	

S#	Num	N	index	effect
	5	5		
1			0	init index
2				true, do 3
3			1	inc index
2				true, do 3
3			2	inc index
2				true, do 3
3			3	inc index
2				true, do 3
3			4	inc index
2				true, do 3
3			5	inc index
2				false, exit

Sequential Search (Sorted)

```
index = 0;                                    /* 1 */
while ((index < N) && (A[index] < Num))       /* 2 */
  index++;                                    /* 3 */
if ((index < N) && (A[index] == Num))         /* 4 */
  printf("Found at %d\n",index);              /* 5 */
else
  printf("Not found\n");                      /* 6 */
```

Can stop either when value found or a value larger than the value being searched for is found

While loop may stop before index reaches N even

Sorted Sequential Search Example

A	N	S#	Num	N	index	effect
1	5		5	5		
4		1			0	init index
6	Num	2				true, do 3
8	5	3			1	inc index
10		2				true, do 3
		3			2	inc index
		2				false, exit
		4				false, do 6
		6				print

Binary Search

Example: when looking up name in phone book, don’t search start to end

Another approach:

- Open book in middle
- Determine if name in left or right half
- Open that half to its middle
- Determine if name in left or right of that half

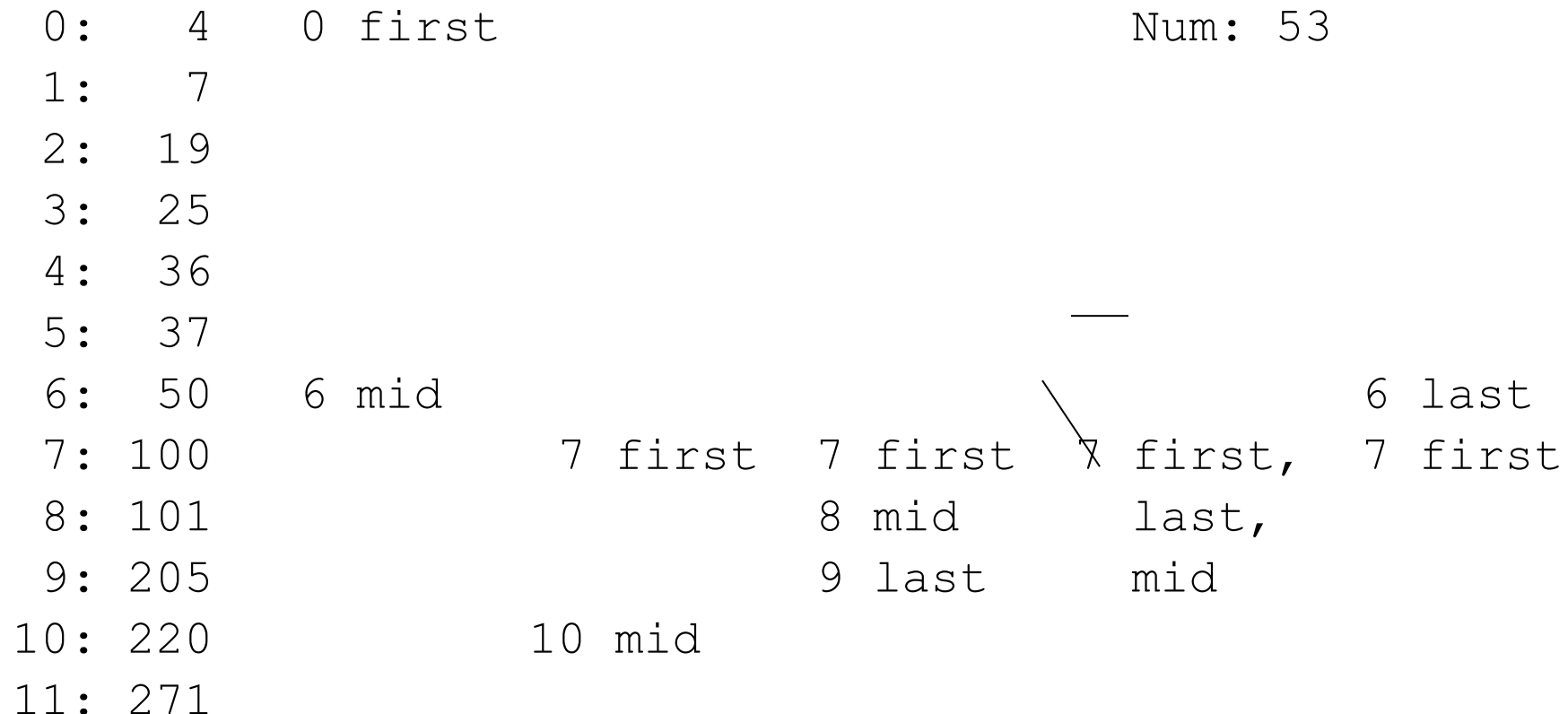
Code for Binary Search

```
first = 0;
last = N - 1;
mid = (first + last) / 2;
while ((A[mid] != num) && (first <= last)) {
  if (A[mid] > num)
    last = mid - 1;
  else
    first = mid + 1;
```

Binary Search Example

```
 0:   4   0 first                          Num: 101
 1:   7
 2:  19
 3:  25
 4:  36
 5:  37
 6:  50   6 mid
 7: 100             7 first  7 first
 8: 101                      8 mid   -- found
 9: 205                      9 last
10: 220           10 mid
11: 271
12: 306
13: 321  13 last  13 last
```

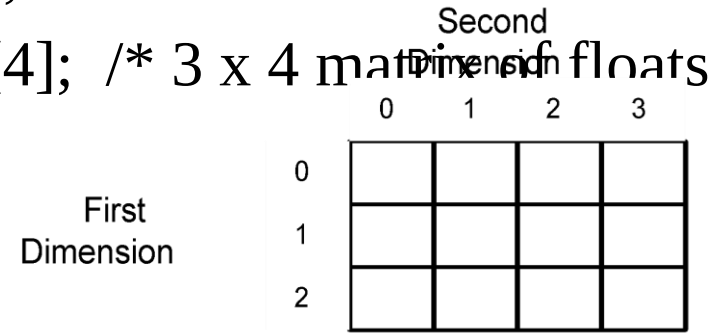
Binary Search Example



2-Dimensional Array Declaration

- Syntax: *BaseType Name*[*IntLit1*][*IntLit2*];
- Examples:

```
int Scores[100][10]; /* 100 x 10 set of scores */
char Maze[5][5];     /* 5 x 5 matrix of chars for Maze */
float FloatM[3][4];  /* 3 x 4 matrix of floats */
```



2D Array Element Reference

- Syntax: *Name*[*intExpr1*][*intExpr2*]
- Expressions are used for the two dimensions in that order
- Values used as subscripts must be legal for each dimension
- Each location referenced can be treated as variable of that type

2D Array Initialization

- 2D arrays can also be initialized
- Syntax:

  BaseType Name[Dim1][Dim2] = { val0, val1, val2, val3, val4, … };

  values are used to initialize first row of matrix, second row, etc.

  BaseType Name[Dim1][Dim2] = {

  { val0-0, val0-1, val0-2, … } /* first row */

Processing 2D Arrays

- Use nested loops to process 2D array

```
Type Name[Dim1][Dim2];

for (J = 0; J < Dim1; J++)
  for (K = 0; K < Dim2; K++)
    process Name[J][K];
```

- Example: print 5x5 Maze

```
for (J = 0; J < 5; J++) {
  for (K = 0; K < 5; K++)
    printf("%c",Maze[J][K];
```

2D Example

```
int Scores[100][10]; /* 10 scores - 100 students */
int J;
int K;

for (J = 0; J < 100; J++) {
  printf(“Enter 10 scores for student %d: “,J);
  for (K = 0; K < 10; K++)
    scanf(“%d”,&(Scores[J][K]));
}
```

Passing 2D Array Parameters

- A single value can be passed as either a value or as a reference parameter
- 2D array may be passed by reference using name, syntax of parameter:

  Type ParamName[][*Dim2*]

  Dim2 must be the same literal constant used in declaring array
- Each row of 2D array may be passed as a

Passing Element of Array

```
int Scores[100][10]; /* 10 scores - 100 students */
int J;
int K;

void readScore(int *score) {
  scanf("%d",score);
}

for (J = 0; J < 100; J++) {
  printf("Enter 10 scores for student %d: ",J);
  for (K = 0; K < 10; K++)
```

Passing Row of Array

```
int Scores[100][10]; /* 10 scores - 100 students */
int J;

void readScore(int *score) {
  scanf("%d",score);
}

void readStudent(int Sscores[], int Snum) {
  int K;

  printf("Enter 10 scores for student %d: ",Snum);
  for (K = 0; K < 10; K++)
    readScore(&(Sscores[K]));
}
```

Passing Entire Array

```
int Scores[100][10]; /* 10 scores - 100 students */

void readScore(int *score) {
  scanf("%d",score);
}

void readStudent(int Sscores[], int Snum) {
  int K;

  printf("Enter 10 scores for student %d: ",Snum);
  for (K = 0; K < 10; K++)
    readScore(&(Sscores[K]));
}

void readStudents(int Ascores[][10]) {
  int J;

  for (J = 0; J < 100; J++)
```

2D Array Example

```
int Scores[100][10]; /* 10 scores - 100 students */

int totalStudent(int Sscores[]) {
  int J;
  int total = 0;
  for (J = 0; J < 10; J++)
    total += Sscores[J];
  return total;
}

float averageStudents(int Ascores[][10]) {
  int J;
  int total = 0;

  for (J = 0; J < 100; J++)
```

Multi-Dimensional Array Declaration

- Syntax:

  BaseType Name[*IntLit1*][*IntLit2*][*IntLit3*]...;
- One constant for each dimension
- Can have as many dimensions as desired
- Examples:

  int Ppoints[100][256][256]; /* 3D array */

  float TimeVolData[100][256][256][256];

Multi-Dim Array Reference

- To refer to an element of multi-dimensional array use one expression for each dimension

  syntax:

  Name[*Expr1*][*Expr2*][*Expr3*] /* 3D */

  Name[*Expr1*][*Expr2*][*Expr3*][*Expr4*] /* 4D */

  etc.

- First expression - first dimension, 2nd expression - 2nd dimension, etc.