BIT 2319 Lecture 1: Review of Data Structures

Philip Apodo Oyier
oyier@itc.jkuat.ac.ke

Learning Outcomes

- Become familiar with some of the fundamental data structures in computer science
- Improve ability to solve problems abstractly
  - Data structures are the building blocks
- Improve ability to analyze your algorithms
  - Prove correctness
  - Gauge (and improve) time complexity
- Become modestly skilled with the C/C++/Java programming

What is Program?

- A Set of Instructions
- Data Structures + Algorithms
- Data Structure = A Container stores Data
- Algorithm = Logic + Control

Overview: System Life Cycle

- Good programmers regard large-scale computer programs as systems that contain many complex interacting parts.
- As systems, these programs undergo a development process called the system life cycle.

Overview: System Life Cycle Contd.

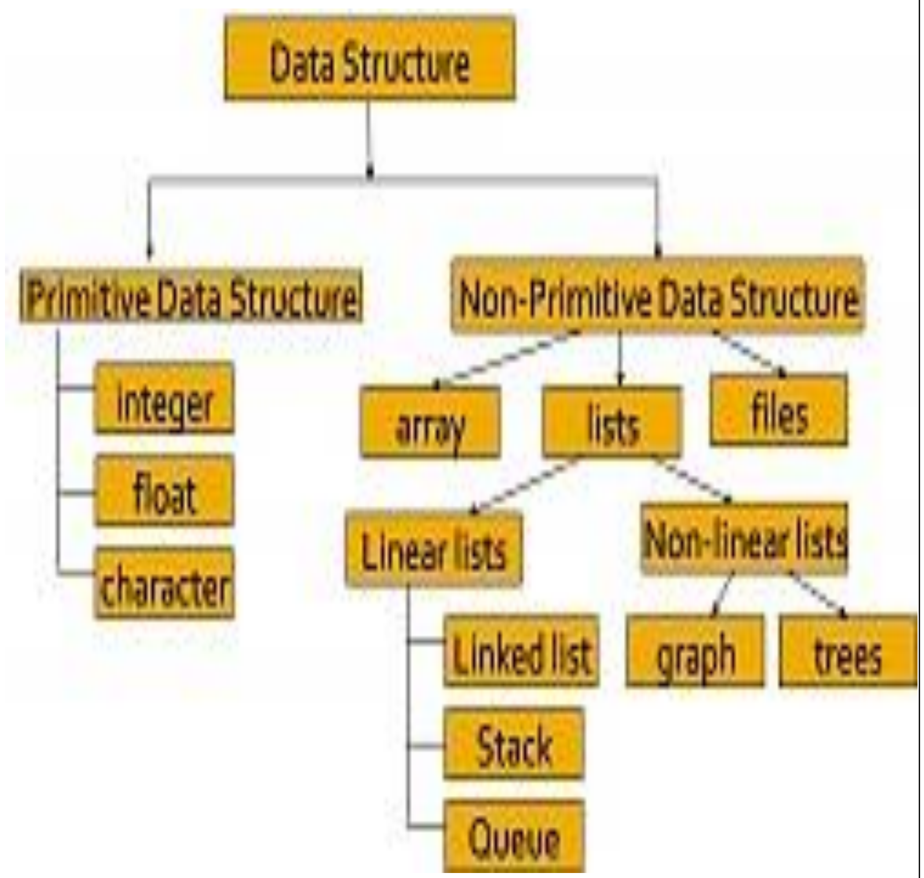
- We consider this cycle as consisting of five phases.
  - Requirements
  - Analysis: bottom-up vs. top-down
  - Design: data objects and operations
  - Refinement and Coding
- Verification
  - Program Proving
  - Testing
  - Debugging

Observation

- All programs manipulate data
  - Programs process, store, display, gather
  - Data can be information, numbers, images, sound
- Each program must decide how to store data
- Choice influences program at every level
  - Execution speed
  - Memory requirements
  - Maintenance (debugging, extending, etc.)

Data Structures

- Data structure is representation of the logical relationship existing between individual elements of data.
- The logical or mathematical model of a particular organization of data is called a ***data structure***.
- A data structure is a way of organizing all data items that considers not only the elements stored but also their relationship to each other.

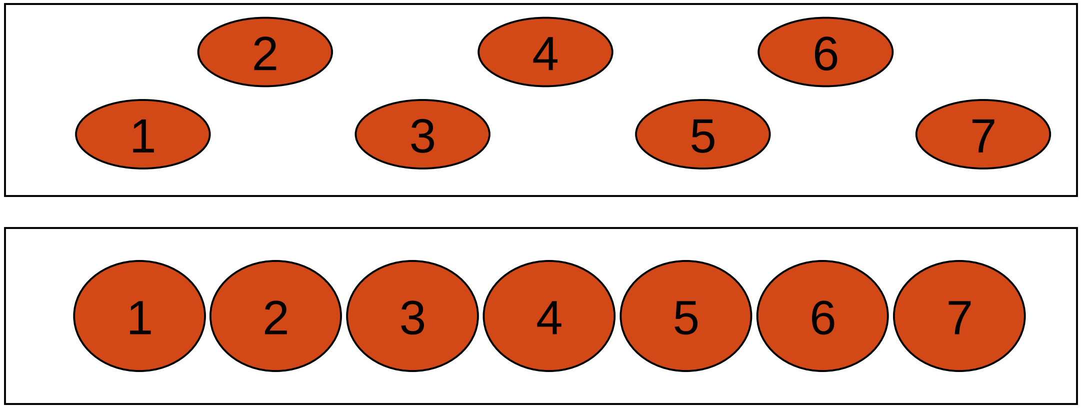


Data Structures Contd.

- A data structure is defined by
  - (1) The logical arrangement of data elements, combined with
  - (2) The set of operations we need to access the elements.
- Atomic Variables
  - Atomic variables can only store one value at a time.
    - int num; float s;
  - A value stored in an atomic variable cannot be subdivided.

What is Algorithm?

- Algorithm:
  - A computable set of steps to achieve a desired result
  - Relationship to Data Structure
    - Example: Find an element



Algorithms + Data Structures = Programs

Algorithms ⟷ Data Structures

What is it all about?

- Solving problems
  - Get me from home to work
  - Balance my checkbook
  - Simulate a jet engine
  - Graduate from JKUAT
- Using a computer to help solve problems
  - Designing programs (architecture, algorithms)
  - Writing programs
  - Verifying programs
  - Documenting programs

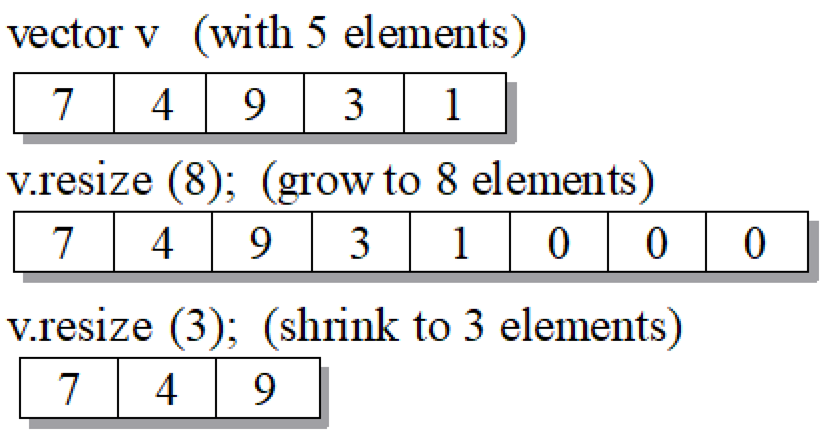
Basic Data Types – the Simplest Data Structure

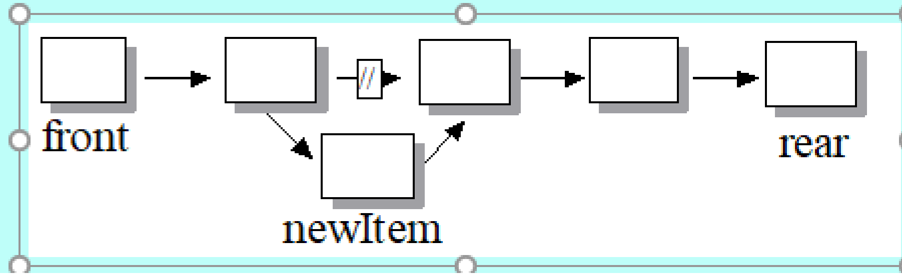
- Basic data types a language supports:
  - Integer, float, double, char, boolean
  - string: usually an array of char supported with library
- A single datum in one of the basic types
- A structure is a combination of the basic types
  - A Publication: code—string, description—string, price–double
- A structure is a combination of basic types and structures
  - An Order item: publication—Publication, quantity–integer, deleteFlag – boolean

Storage Container

- For storing multiple occurrences of a structure
- Contiguous structures:
  - Array – supported by a language but needs care of array size, overflow
  - Vector – a structure to allow handling of its size and overflow "automatically"
- Linked List: allow data connected by "links" to save space which may be wasted in an array/vector.
- Combination of vector and linked list

Examples of Storage Containers





Notes on the Basic Storage Containers

- Vector and list allow data to be stored in different ways but not restricted to any order, or any operations, e.g.,
  - Data can be ordered in any sequence, even though searching may prefer a sorted one.
  - Operations support inserting an element in between elements of a vector, even though it may involve a lot of "shift" operations.

Data Structures and Algorithms

- Algorithm
  - Outline, the essence of a computational procedure, step-by-step instructions
  - A high level, language independent description of a step-by-step process for solving a problem
- Program
  - An implementation of an algorithm in some programming language
- Data structure
  - **Organization** of data needed to solve the problem
  - A set of algorithms which implement an ADT

Why so many Data Structures?

- Ideal data structure:
  - fast, elegant, memory efficient
- Generates tensions:
  - time vs. space
  - performance vs. elegance
  - generality vs. simplicity
  - one operation's performance vs. another's
- Dictionary ADT
  - list
  - binary search tree
  - AVL tree
  - Splay tree
  - Red-Black tree
  - hash table

Code Implementation

- Theoretically
  - Abstract base class describes ADT
  - Inherited implementations implement data structures
  - Can change data structures transparently (to client code)
- Practice
  - Different implementations sometimes suggest different interfaces (generality vs. simplicity)
  - Performance of a data structure may influence form of client code (time vs. space, one operation vs. another)

ADT Presentation Algorithm

- Present an ADT
- Motivate with some applications
- Repeat until browned entirely through
  - Develop a data structure for the ADT
  - Analyze its properties
  - Efficiency
  - Correctness
  - Limitations
  - Ease of programming
- Contrast data structure's strengths and weaknesses
  - Understand when to use each one

Algorithm Strategies

- There are countless algorithms
- Strategies
  - Greedy
  - Divide and Conquer
  - Dynamic Programming
  - Exhaustive Search

Overall Picture

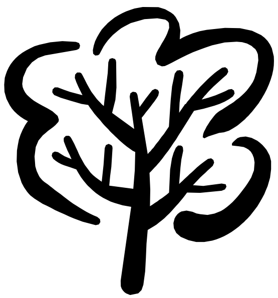
Data Structure and Algorithm Design Goals

Correctness

Efficiency

Implementation Goals

Robustness



Adaptability

Reusability

Terminology

- Abstract Data Type (ADT)
  - Mathematical description of an object with set of operations on the object. Useful building block.
- Algorithm
  - A high level, language independent, description of a step-by-step process
- Data structure
  - A specific family of algorithms for implementing an abstract data type.
- Implementation of data structure
  - A specific implementation in a specific language

Data Abstraction

- Data Type
  - A data type is a collection of objects and a set of operations that act on those objects.
  - For example, the data type int consists of the objects {0, +1, -1, +2, -2, …, INT_MAX, INT_MIN} and the operations +, -, *, /, and %.
- The data types of C/C++
  - The basic data types: char, int, float and double
  - The group data types: array and struct
  - The pointer data type
  - The user-defined types

Data Abstraction: Specification vs. Implementation

- Specification vs. Implementation
  - An ADT is implementation independent
  - Operation specification
    - Function name
    - The types of arguments
    - The type of the results
  - The functions of a data type can be classify into several categories:
    - Creator / constructor
    - Transformers
    - Observers / reporters

Data abstraction Example: Abstract Data Type Natural_Number

structure *Natural_Number* is

objects: an ordered subrange of the integers starting at zero and ending at the maximum integer (*INT_MAX*) on the computer

functions:

for all $x, y \in$ *Nat_Number*; *TRUE*, *FALSE* $\in$ *Boolean*
and where +, −, <, and == are the usual integer operations

Nat_No Zero( )	::=	0
Boolean Is_Zero(x)	::=	**if** (x) **return** *FALSE* **else return** *TRUE*
Nat_No Add(x, y)	::=	**if** (($x + y$) <= *INT_MAX*) **return** $x + y$ **else return** *INT_MAX*
Boolean Equal(x, y)	::=	**if** ($x == y$) **return** *TRUE* **else return** *FALSE*
Nat_No Successor(x)	::=	**if** ($x ==$ *INT_MAX*) **return** x **else return** $x + 1$
Nat_No Subtract(x, y)	::=	**if** ($x < y$) **return** 0 **else return** $x - y$

end *Natural_Number*

Structure 1.1: Abstract data type *Natural_Number*

The Abstract Data Type

- An ADT consists of a data declaration packaged together with the operations that are meaningful on the data while embodying the structured principles of encapsulation and data hiding.
- The basic parts of an ADT.
  - Atomic and Composite Data
  - Data Type
  - Data Structure
  - Abstract Data Type

Type	Values	Operations
integer	$-\infty$, … , -2, -1, 0, 1, 2,…, ∞	*, +, -, %, /, ++, --, …
floating point	$-\infty$, … , 0.0, … , ∞	*, +, -, /, …
character	\0, …, 'A', 'B', … , 'a', 'b', … , ~	<, >, …

TABLE 1-1 Three Data Types

Abstract Data Type

- ADT users are NOT concerned with how the task is done but rather what it can do.
- An abstract data type is a **data declaration** packaged together with the **operations** that are meaningful for the data type.
- We **encapsulate** the data and the operations on the data, and then hide them from the user.
- All references to and manipulation of the data in a data structure are handled through defined interfaces to the structure.

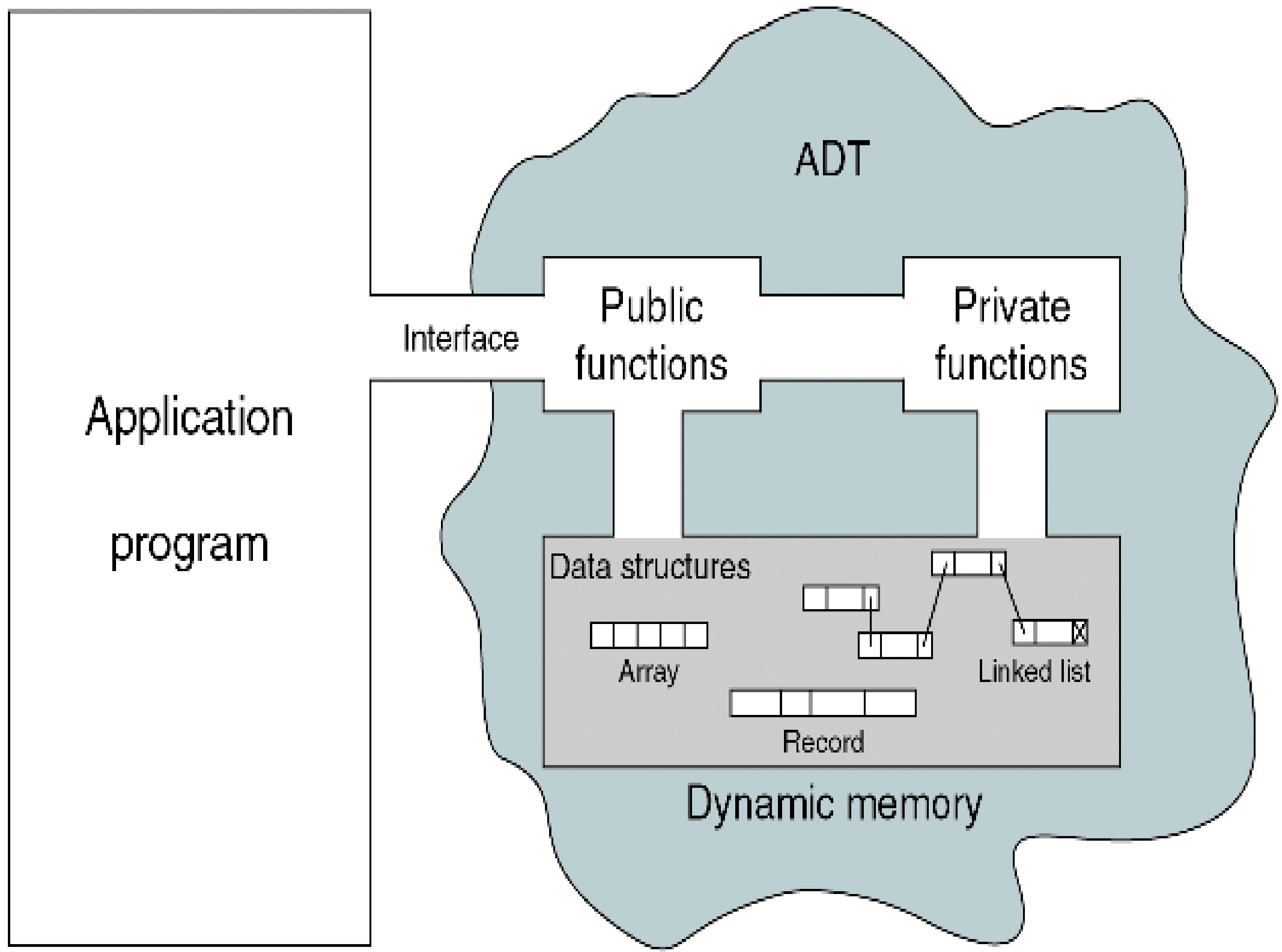


FIGURE 1-2 Abstract Data Type Model

Data Structure

- Aggregation of atomic and composite data into a set with defined relationships.
- Structure refers to a set of rules that hold the data together.
- A combination of elements in which each is either a data type or another data structure.
- A set of associations of relationship involving combined elements.
- Example:

Array	Record
Homogeneous sequence of data or data types known as elements	Heterogeneous combination of data into a single structure with an identified key
Position association among the elements	No association

TABLE 1-2 Data Structure Examples

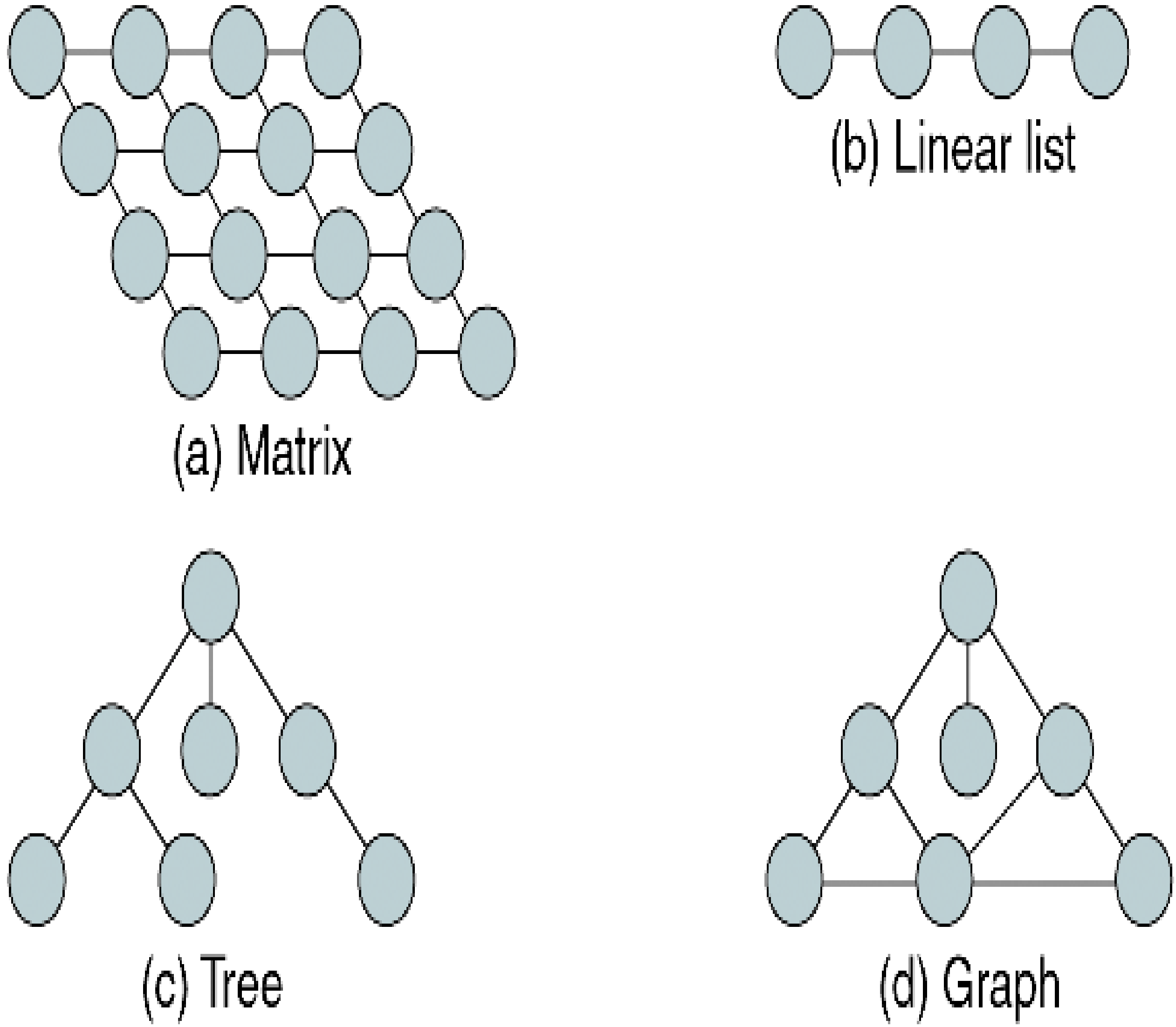


FIGURE 1-1 Some Data Structures

ADT Implementations

- There are two basic structures we can use to implement an ADT list: arrays and linked lists.
  - Array Implementation
  - Linked List Implementation

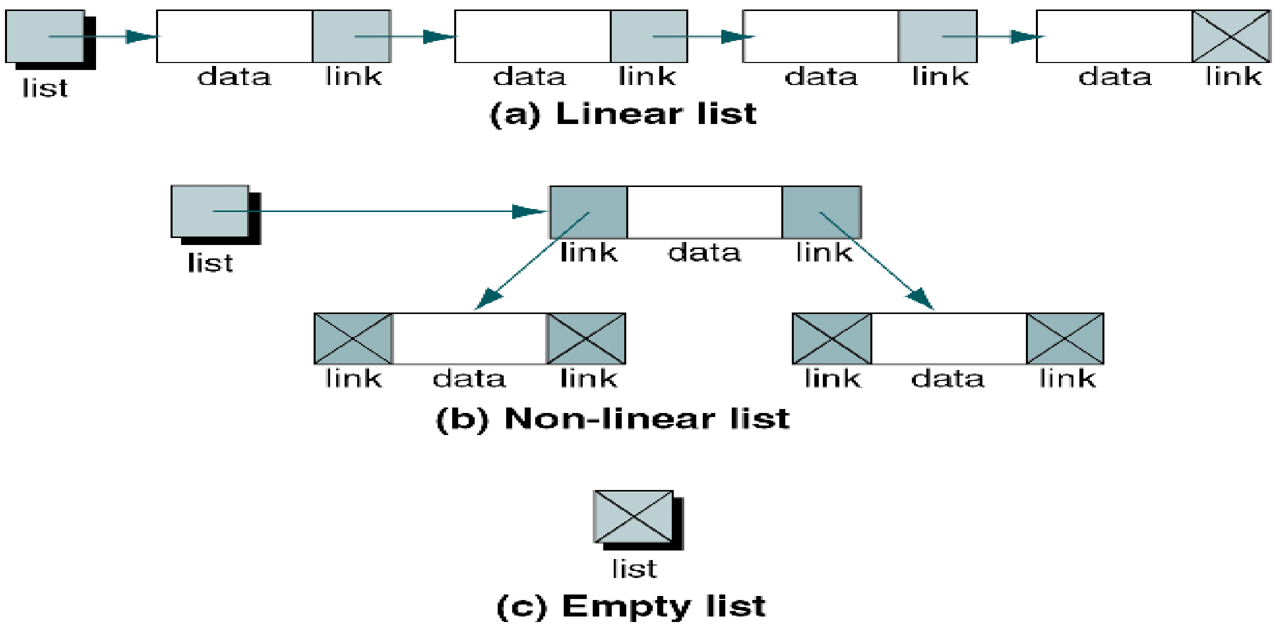


FIGURE 1-3 Linked Lists

ADT Implementations Contd.

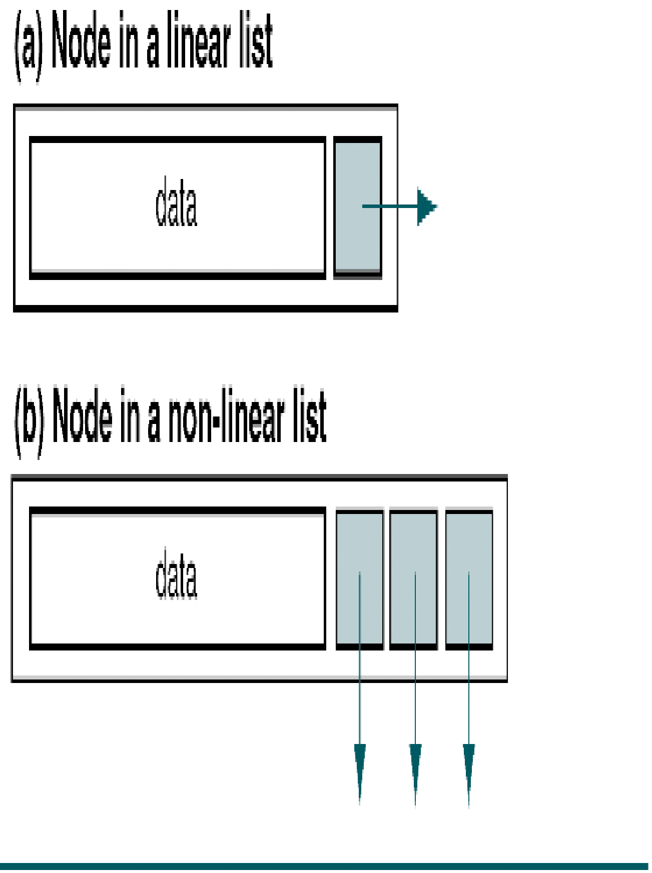


FIGURE 1-4 Nodes

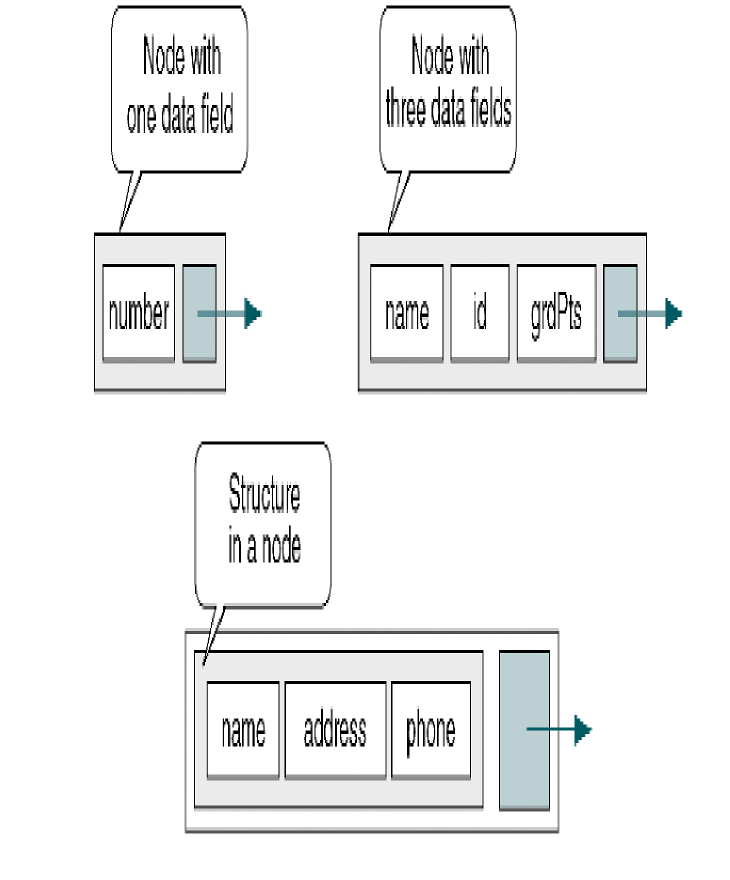


FIGURE 1-5 Linked List Node Structures

Generic Code for ADT

- C/C++ tools that are required to implement an ADT.
  - Pointer to Void
  - Pointer to Function

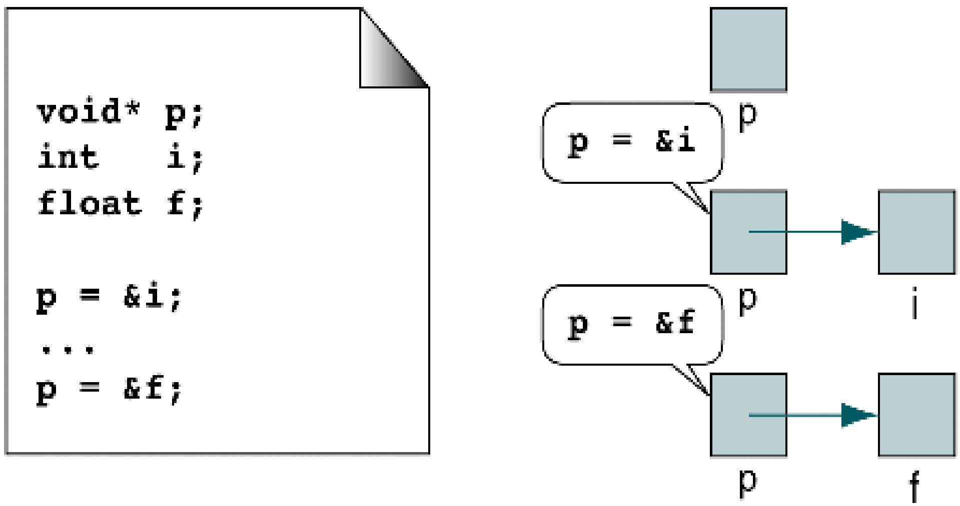


FIGURE 1-7 Pointers for Program 1-1

PROGRAM 1-1 Demonstrate Pointer to *void*

```c
/* Demonstrate pointer to void.
      Written by:
      Date:
*/
#include <stdio.h>

int main ()
{
// Local Definitions
   void* p;
   int   i = 7 ;
   float f = 23.5;

// Statements
   p = &i;
   printf ("i contains: %d\n", *((int*)p) );

   p = &f;
   printf ("f contains: %f\n", *((float*)p));

   return 0;
}  // main
```

```
Results:
i contains 7
f contains 23.500000
```

Generic Code for ADT

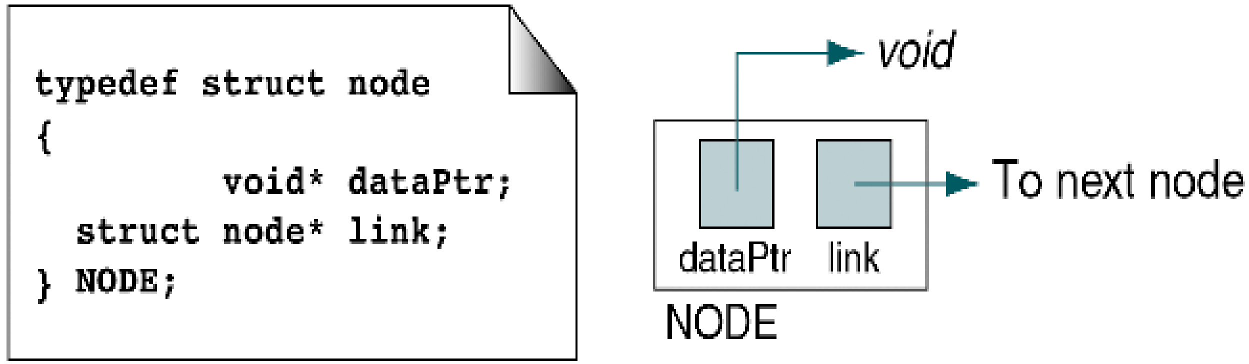


FIGURE 1-8 Pointer to Node

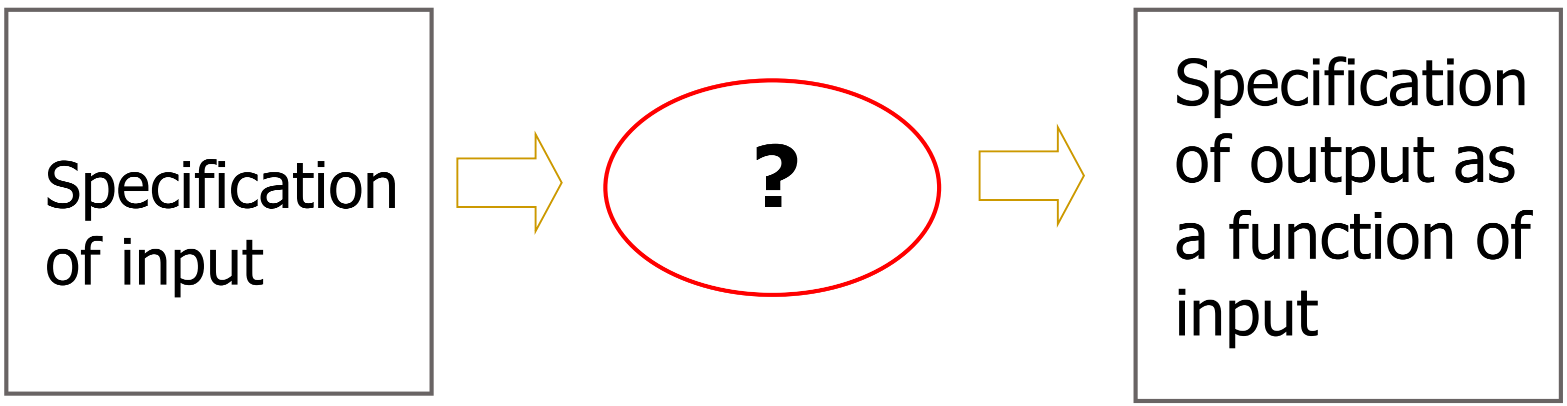
```
/* Header file for create node structure.
      Written by:
      Date:
*/
typedef struct node
{
            void* dataPtr;
   struct node* link;
} NODE;

```

PROGRAM 1-2 Create Node Header File

```
/* ==================== createNode ====================
   Creates a node in dynamic memory and stores data
   pointer in it.
      Pre  itemPtr is pointer to data to be stored.
      Post node created and its address returned.
*/
NODE* createNode (void* itemPtr)
{
   NODE* nodePtr;
   nodePtr = (NODE*) malloc (sizeof (NODE));
   nodePtr->dataPtr = itemPtr;
   nodePtr->link    = NULL;
   return nodePtr;
} // createNode
```

Algorithmic Problem



- Infinite number of input *instances* satisfying the specification.
- For example:
  - A sorted, non-decreasing sequence of natural numbers. The sequence is of non-zero, finite length:
    - 1, 20, 908, 909, 100000, 1000000000.
    - 3.

Algorithmic Solution

Input instance, adhering to the specification

Algorithm

Output related to the input as required

- Algorithm describes actions on the input instance
- Infinitely many correct algorithms for the same algorithmic problem

Algorithm Specification

- An algorithm is a finite set of instructions that accomplishes a particular task.
- Criteria
  - Input: zero or more quantities that are externally supplied
  - Output: at least one quantity is produced
  - Definiteness: clear and unambiguous
  - Finiteness: terminate after a finite number of steps
  - Effectiveness: instruction is basic enough to be carried out
- A program does not have to satisfy the finiteness criteria.

Algorithm Specification : Representation

- Representation
  - A natural language, like English
  - A graphic, like flowcharts or UML diagrams
  - A computer language, like C or C++ or Java or VB
- Algorithms + Data structures = Programs
- Sequential search vs. Binary search

Pseudocode

- Pseudocode is an English-like representation of the algorithm logic.
- It consists of an extended version of the basic algorithmic constructs: sequence, selection, and iteration.
  - Algorithm Header
  - Purpose, Condition, and Return
  - Statement Numbers
  - Variables
  - Statement Constructs
  - Algorithm Analysis

ALGORITHM 1-1 Example of Pseudocode

```
Algorithm sample (pageNumber)
This algorithm reads a file and prints a report.
   Pre     pageNumber passed by reference
   Post    Report Printed
           pageNumber contains number of pages in report
   Return Number of lines printed
1  loop (not end of file)
   1  read file
   2  if (full page)
      1  increment page number
      2  write page heading
   3  end if
   4  write report line
   5  increment line count
2  end loop
3  return line count
end sample
```

ALGORITHM 1-2 Print Deviation from Mean for Series

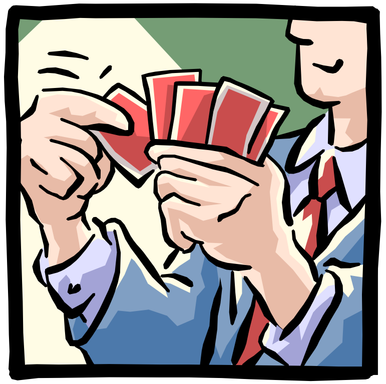
```
Algorithm deviation
   Pre     nothing
   Post    average and numbers with their deviation printed
1  loop (not end of file)
   1   read number into array
   2   add number to total
   3   increment count
2  end loop
3  set average to total / count
4  print average
5  loop (not end of array)
   1   set devFromAve to array element - average
   2   print array element and devFromAve
6  end loop
end deviation
```

Example: Sorting

INPUT
sequence of numbers

$a_1, a_2, a_3, \ldots, a_n$

2 5 4 10 7



OUTPUT
a permutation of the sequence of numbers

$b_1, b_2, b_3, \ldots, b_n$

2 4 5 7 10

Correctness
For any given input the algorithm halts with the output:
- $b_1 < b_2 < b_3 < \ldots < b_n$
- $b_1, b_2, b_3, \ldots, b_n$ is a permutation of $a_1, a_2, a_3, \ldots, a_n$

Running time
Depends on
- number of elements (n)
- how (partially) sorted they are
- algorithm

Algorithm Specification Example

- **Example 1.1 [*Selection sort*]:**
  - From those integers that are currently unsorted, find the smallest and place it next in the sorted list.

i	[0]	[1]	[2]	[3]	[4]
-	30	10	50	40	20
0	10	30	50	40	20
1	10	20	40	50	30
2	10	20	30	40	50
3	10	20	30	40	50

```
for (i = 0; i < n; i++) {
  Examine list[i] to list[n-1] and suppose that the
  smallest integer is  at list[min];

  Interchange list[i] and list[min];
}
```

Program 1.1: Selection sort algorithm

A complete Program for Selection Sort

```
#include <stdio.h>
#include <math.h>
#define MAX_SIZE 101
#define SWAP(x,y,t) ((t) = (x), (x)= (y), (y) = (t))
void sort(int [],int); /*selection sort */
void main(void)
{
   int i,n;
   int list[MAX_SIZE];
   printf("Enter the number of numbers to generate: ");
   scanf("%d",&n);
   if( n < 1 || n > MAX_SIZE) {
     fprintf(stderr, "Improper value of n\n");
     exit(1);
   }
   for (i = 0; i < n; i++) {/*randomly generate numbers*/
      list[i] = rand() % 1000;
      printf("%d  ",list[i]);
   }
   sort(list,n);
   printf("\n Sorted array:\n ");
   for (i = 0; i < n; i++) /* print out sorted numbers */
      printf("%d  ",list[i]);
   printf("\n");
}
void sort(int list[],int n)
{
   int i, j, min, temp;
   for (i = 0; i < n-1; i++)  {
      min = i;
      for (j = i+1; j < n; j++)
         if (list[j] < list[min])
            min = j;
      SWAP(list[i],list[min],temp);
   }
}
```

Program 1.3: Selection sort

Algorithm Specification: Recursive Algorithms

- Beginning programmer view a function as something that is invoked (called) by another function
  - It executes its code and then returns control to the calling function.
- This perspective ignores the fact that functions can call themselves (direct recursion).
- They may call other functions that invoke the calling function again (indirect recursion).
  - Extremely powerful
  - Frequently allow us to express an otherwise complex process in very clear term
- We should express a recursive algorithm when the problem itself is defined recursively

Algorithm Specification: Recursive Binary Search

```
int binsearch(int list[], int searchnum, int left,
                                          int right)
{
/* search list[0] <= list[1] <=  · · ·  <= list[n-1] for
searchnum. Return its position if found. Otherwise
return -1 */
   int middle;
   if (left <= right) {
      middle = (left + right)/2;
      switch (COMPARE(list[middle], searchnum)) {
         case -1: return
            binsearch(list, searchnum, middle + 1, right);
         case 0 : return middle;
         case 1 : return
            binsearch(list, searchnum, left, middle - 1);
      }
   }
   return -1;
}
```

Program 1.7: Recursive implementation of binary search

Algorithm Analysis: Why?

- Criteria
  - Is it correct i.e., Does the algorithm do what is intended?
  - Is it readable?
- Performance (machine independent)
  - Time complexity: What is the running time of the algorithm.
  - Space complexity: How much storage does it consume.
- Efficiency as a function of input size:
  - Number of data elements (numbers, points)
  - A number of bits in an input number
- Different algorithms may correctly solve a given task
  - Which should I use?

Algorithm Efficiency

- To design and implement algorithms, programmers must have a basic understanding of what constitutes good, efficient algorithms.
- Linear Loops
  - Efficiency is a function of the number of instructions.
  - Loop update either adds or subtracts.
- Logarithmic Loops
  - The controlling variable is either multiplied or divided in each iteration.
  - The number of iteration is a function of the multiplier or divisor.
- Nested Loops
  - The number of iterations is the total number which is the product of the number of iterations in the inner loop and number of iterations in the outer loop.
- Big-O Notation
  - Not concerned with exact measurement of efficiency but with the magnitude.
  - A dominant factor determines the magnitude.

Performance analysis: Time Complexity

- Time Complexity: $T(P)=C+T_P(I)$
  - The time, $T(P)$, taken by a program, P, is the sum of its compile time C and its run (or execution) time, $T_P(I)$
- Fixed time requirements
  - Compile time (C), independent of instance characteristics
- Variable time requirements
  - Run (execution) time T_P
  - $T_P(n)=c_aADD(n)+c_sSUB(n)+c_lLDA(n)+c_{st}STA(n)$

Performance analysis: Time Complexity Contd.

- A program step is a syntactically or semantically meaningful program segment whose execution time is independent of the instance characteristics.
- Example (Regard as the same unit machine independent)
  - abc = a + b + b * c + (a + b - c) / (a + b) + 4.0
  - abc = a + b + c
- Methods to compute the step count
  - Introduce variable count into programs
  - Tabular method
    - Determine the total number of steps contributed by each statement step per execution × frequency
    - Add up the contribution of all statements

Performance analysis: Time Complexity Contd.

- Iterative summing of a list of numbers
- **Example:** Program with count statements

```
float sum(float list[ ], int n) {
   float tempsum = 0; count++; /* for assignment */
   int i;
   for (i = 0; i < n; i++) {
        count++;            /*for the for loop */
        tempsum += list[i]; count++;  /* for assignment */
   }
   count++;        /* last execution of for */
   count++;        /* for return */
  return tempsum;
}
```

2n + 3 steps

Performance analysis: Time Complexity Contd.

- Tabular Method
- *Figure below: Step count table for Program
- Iterative function to sum a list of numbers

Steps/execution

Statement	s/e	Frequency	Total steps
float sum(float list[ ], int n)	0	0	0
{	0	0	0
float tempsum = 0;	1	1	1
int i;	0	0	0
for(i=0; i <n; i++)	1	n+1	n+1
tempsum += list[i];	1	n	n
return tempsum;	1	1	1
}	0	0	0
Total			2n+3

Performance analysis: Time Complexity Contd.

- Recursive summing of a list of numbers
- Program with count statements added

```
float rsum(float list[ ], int n) {
        count++;      /*for if conditional */
        if (n) {
                count++;  /* for return and rsum invocation*/
                return rsum(list, n-1) + list[n-1];
        }
        count++;
        return list[0];
}
```

2n+2 steps

Performance analysis: Time Complexity Contd.

- Step count table for recursive summing function

Statement	s/e	Frequency	Total steps
float rsum(float list[ ], int n)	0	0	0
{	0	0	0
if (n)	1	n+1	n+1
return rsum(list, n-1)+list[n-1];	1	n	n
return list[0];	1	1	1
}	0	0	0
Total			2n+2

Performance Analysis Asymptotic notation (O, Ω, Θ)

- **Definition**: [Big "oh"]
  - f(*n*) = O(g(*n*)) iff there exist positive constants c and n_0 such that f(*n*) ≤ cg(*n*) for all n, n ≥ n_0.
- **Definition:** [Omega]
  - $f(n) = \Omega(g(n))$ (read as "*f* of *n* is omega of *g* of *n*") iff there exist positive constants *c* and n_0 such that $f(n) \geq cg(n)$ for all n, $n \geq n_0$.
- **Definition:** [Theta]
  - $f(n) = \Theta(g(n))$ (read as "*f* of *n* is theta of *g* of *n*") iff there exist positive constants c_1, c_2, and n_0 such that $c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all n, $n \geq n_0$.

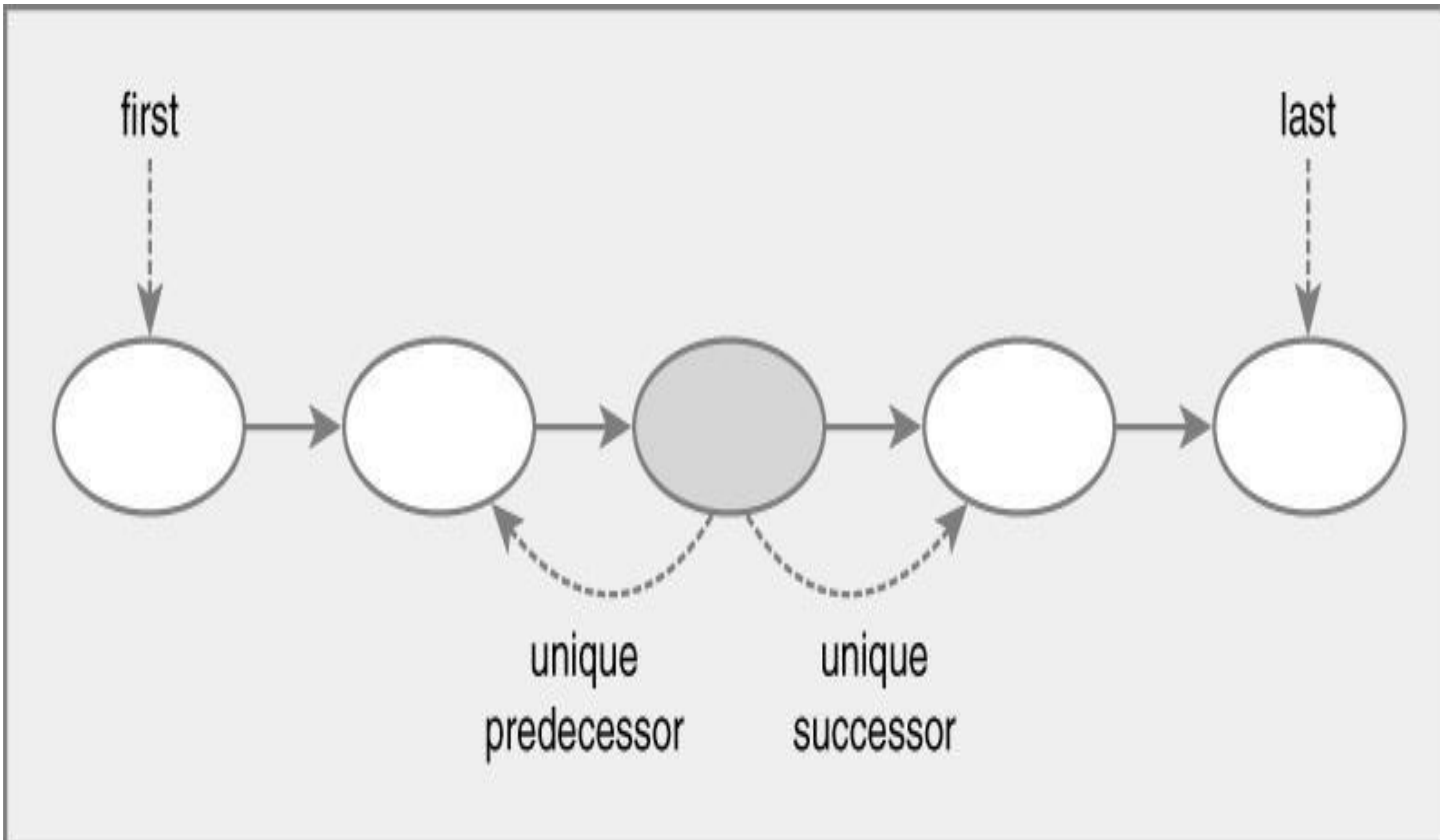
Which Data Structure or Algorithm is Better?

- Must Meet Requirement
- High Performance
- Low RAM footprint
- Easy to implement
  - Encapsulated

Linear Data Structures

- A data structure is said to be linear if its elements form a sequence or a linear list.
  - Arrays
  - Linked Lists
  - Stacks, Queues
- A ***one:one*** relationship between elements in the collection.
  - Assuming the structure is not empty, there is a first and a last element.
  - Every element ***except the first*** has a unique predecessor.
  - Every element ***except the last*** has a unique successor.

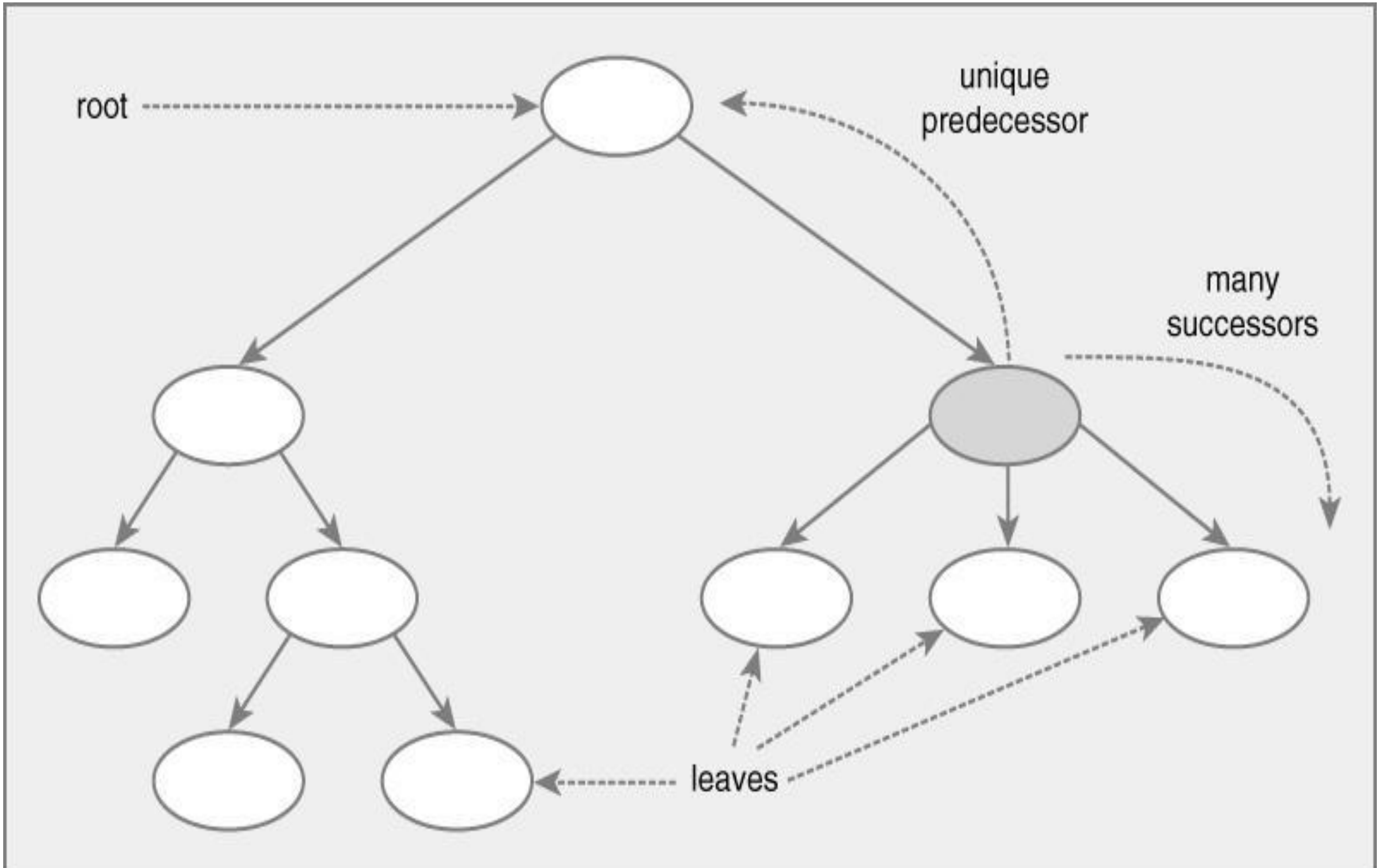
General model of a linear data structure



Hierarchical Data Structures

- Hierarchical Data Structures
  - A ***one:many*** relationship between elements in the collection.
    - Assuming the structure is not empty, there is a unique element called the ***root.***
    - There may be ***zero to many*** terminating nodes called leaves.
    - Nodes that are neither ***roots nor leaves*** are called ***internal***.

General model of a hierarchical data structure



Hierarchical Data Structures contd.

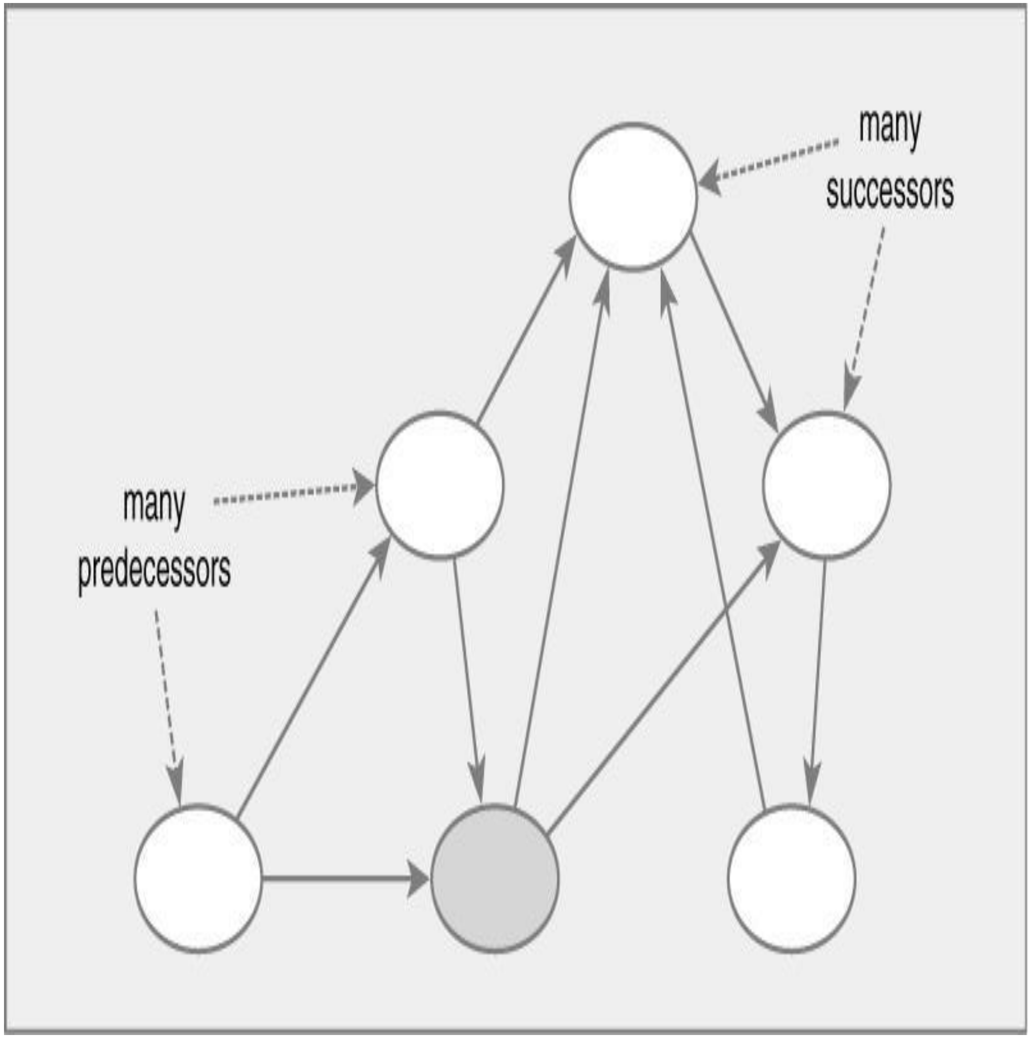
- Every element except the root has a unique ***predecessor***.
- Every element except a leaf has a one or more ***successors***.
- An ***internal*** node has ***exactly one predecessor*** and ***one or more successors.***
- There is more than one way to traverse a hierarchical data structure.
- Generally called ***trees***, they are very important and widely used.

Hierarchical Data Structures contd.

- A few types are; Generalized Trees, Binary Trees, Binary Search Trees, AVL Trees (balanced binary search trees), Splay Trees, B Trees, & P Trees.
- Similar to linear data types, the basic structure is the same.
- Each version has different rules and operations.

Graph Data Structures

- A ***many:many*** relationship between elements in the collection.
- An element (***E***) in graph can be ***connected*** arbitrarily to any other element in the graph, (including itself).
- Conversely, any number of elements can be connected to ***E.***



General model of a graph data structure

Linear Data Structures

- Traversal through a liner data structure is called ***iteration***.
- The basic structures are the same.
- The operations and restrictions are different.

Operations on Linear Data Structure

- ***Traversal:*** Travel through the data structure.
- ***Search:*** Traversal through the data structure for a given element.
- ***Insertion:*** Adding new elements to the data structure.
- ***Deletion:*** Removing an element from the data structure.
- ***Sorting***: Arranging the elements in some type of order.
- ***Merging:*** Combining two similar data structures into one.

Queue ADT

Queue ADT

- Queue operations
  - create
  - destroy
  - enqueue
  - dequeue
  - is_empty

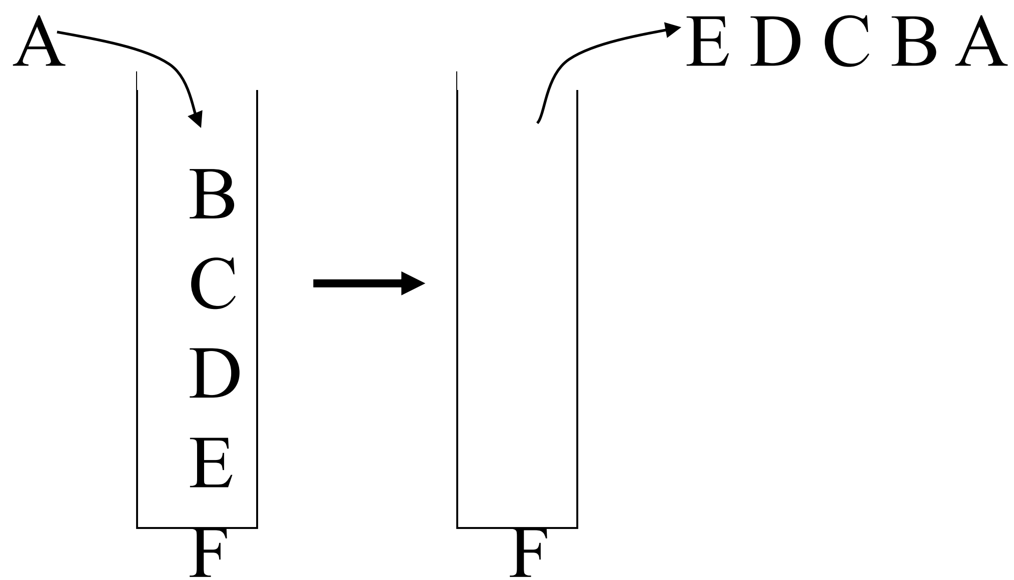
- Queue property: if x is enQed before y is enQed, then x will be deQed before y is deQed
  - FIFO: First In First Out

Applications of the Queue

- Hold jobs for a printer
- Store packets on network routers
- Hold memory "freelists"
- Make waitlists fair
- Breadth first search

LIFO Stack ADT

- Stack operations
  - create
  - destroy
  - push
  - pop
  - top
  - is_empty

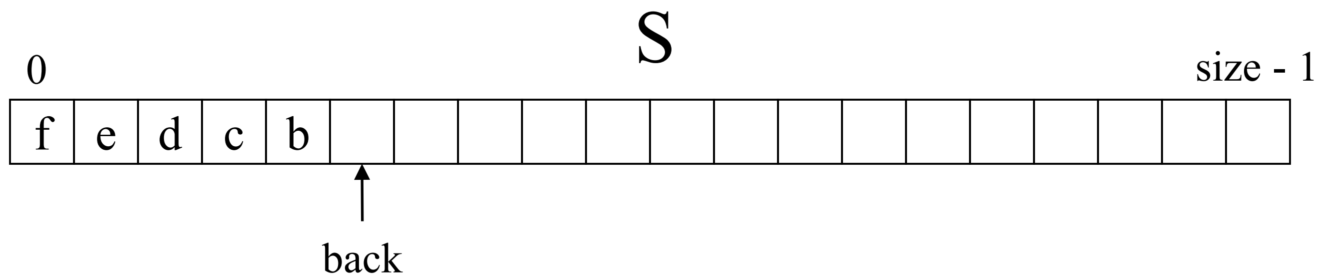


- Stack property: if x is on the stack before y is pushed, then x will be popped after y is popped
  - LIFO: Last In First Out

Stacks in Practice

- Function call stack
- Removing recursion
- Balancing symbols (parentheses)
- Evaluating Reverse Polish Notation
- Depth first search

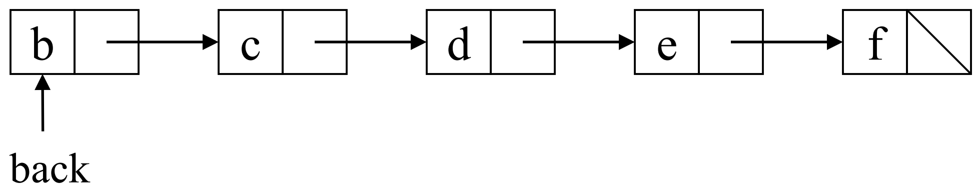
Array Stack Data Structure




```
void push(Object x) {
    assert(!is_full())
    S[back] = x
    back++
}
Object top() {
    assert(!is_empty())
    return S[back - 1]
}
```

```
Object pop() {
    back--
    return S[back]
}
bool is_full() {
    return back == size
}
```

Linked List Stack Data Structure




```
void push(Object x) {
    temp = back
    back = new Node(x)
    back->next = temp
}
Object top() {
    assert(!is_empty())
    return back->data
}
```

```
Object pop() {
    assert(!is_empty())
    return_data = back->data
    temp = back
    back = back->next
    return return_data
}
```

Data Structures you should already know

- Arrays
- Linked lists
- Trees
- Queues
- Stacks

Proof by Induction

- **Basis Step:**
  - The algorithm is correct for the base case (e.g. $n=0$) by inspection.
- **Inductive Hypothesis ($n=k$):**
  - Assume that the algorithm works correctly for the first k cases, for any k.
- **Inductive Step ($n=k+1$):**
  - Given the hypothesis above, show that the $k+1$ case will be calculated correctly.

Program Correctness by Induction

- **Basis Step:** sum(v,0) = 0. ✓
- **Inductive Hypothesis (n=k):**
  - Assume sum(v,k) correctly returns sum of first k elements of v, i.e. **`v[0]+v[1]+`**…**`+v[k-1]`**
- **Inductive Step (n=k+1):**
  - sum(v,n) returns **`v[k]+sum(v,k)`** which is the sum of first k+1 elements of v. ✓