DS - Module

Team Emertxe

Chapter 1: Introduction

Abstract Data Types:
ADT

- A set of data values and associated operations that are precisely specified independent of any particular implementation.
- Example: stack, queue, priority queue.

DataStructures

- The term data structure refers to a scheme for organizing related pieces of Information.
- The basic types of data structures include: files, lists, arrays,records, trees, tables.
- A data structure is the concrete implementation of that type, specifying how much memory is required and, crucially, how fast the execution of each operation will be

Timing

- Every time we run the program we need to estimate how long a program will run since we are going to have different input values so the running time will vary.
- The worst case running time represents the maximum running time possible for all input values.
- We call the worst case timing big Oh written O(n).The n represents the worst case execution time units.

Complexities

- Simple programming statement
- Example:

```
k++;
```

- Complexity : O(1)
- Simple programming statements are considered 1 time unit.

Complexities

- Linear ***for*** loops
- Example:

```
k = 0;
for(i = 0; i < n; i++)
   k++;
```

- Complexity : O(n)
- ***for*** loops are considered n time units because they will repeat a programming statement n times.
- The term linear means the for loop increments or decrements by 1

Complexities

- Non Linear loops
- Example:

```
k=0;
for(i=n; i>0; i=i/2)
    k++;
```

- Complexity : O(log n)
- For every iteration of the loop counter i will divide by 2.
- If i starts is at 16 then then successive i's would be 16, 8, 4, 2, 1. The final value of k would be 4. Non linear loops are logarithmic.
- The timing here is definitely $\log_2 n$ because $2^4 = 16$. Can also works for multiplication.

Complexities

- Nested ***for*** loops
- Example:

```
k=0;
for(i=0; i<n; i++)
   for(j=0; j<n; j++)
     k++;
```

- Complexity : $O(n^2)$
- Nested for loops are considered n^2 time units because they represent a loop executing inside another loop.

Complexities

- Sequential ***for*** loops
- Example:

```
k=0;
for(i=0; i<n; i++)
   k++;
k=0;
for(j=0; j<n; j++)
   k++;
```

- Complexity : O(n)
- Sequential for loops are not related and loop independently of each other.

Complexities

- Loops with non-linear inner loops
- Example:

```
k=0;
for(i=0;i<n;i++)
   for(j=i; j>0; j=j/2)
      k++;
```

- Complexity : O(n log n)

- The outer loop is O(n) since it increments linear.
- The inner loop is O(n log n) and is non-linear because decrements by dividing by 2.
- The final worst case timing is: O(n) * O(log n) = O(n log n)

Complexities

- Inner loop incrementer initialized to outer loop incrementer
- Example:

```
k=0;
for(i=0;i<n;i++)
   for(j=i;j<n;j++)
      k++;
```

- Complexity : O(n^2)
- In this situation we calculate the worst case timing using both loops.
- For every i loop and for start of the inner loop j will be n-1 , n-2, n-3.

Complexities

- Power Loops
- Example:

```
k=0;
for(i=1;i<=n;i=i*2)
   for(j=1;j<=i;j++)
      k++;
```

- Complexity : O(2^n)

- To calculate worst case timing we need to combine the results of both loops.
- For every iteration of the loop counter i will multiply by 2.
- The values for j will be 1, 2, 4, 8, 16 and k will be the sum of these numbers 31 which is $2^n - 1$.

Complexities

- If-else constructs
- Example:

```
/* O(n) */
if (x == 5)
{
     k=0;
     for(i=0; i<n; i++)
       k++;
 }
/* O(n2) */
else
{
    k=0;
    for(i=0;i<n;i++)
        for(j=i; j>0; j=j/2)
            k++;
}
```

- Complexity : $O(n^2)$

Complexities

- Recursive
- Example:

```
int f(int n)
{
      if(n == 0)
            return 0;
       else
            return f(n-1) + n
 }
```

- Complexity : O(n)

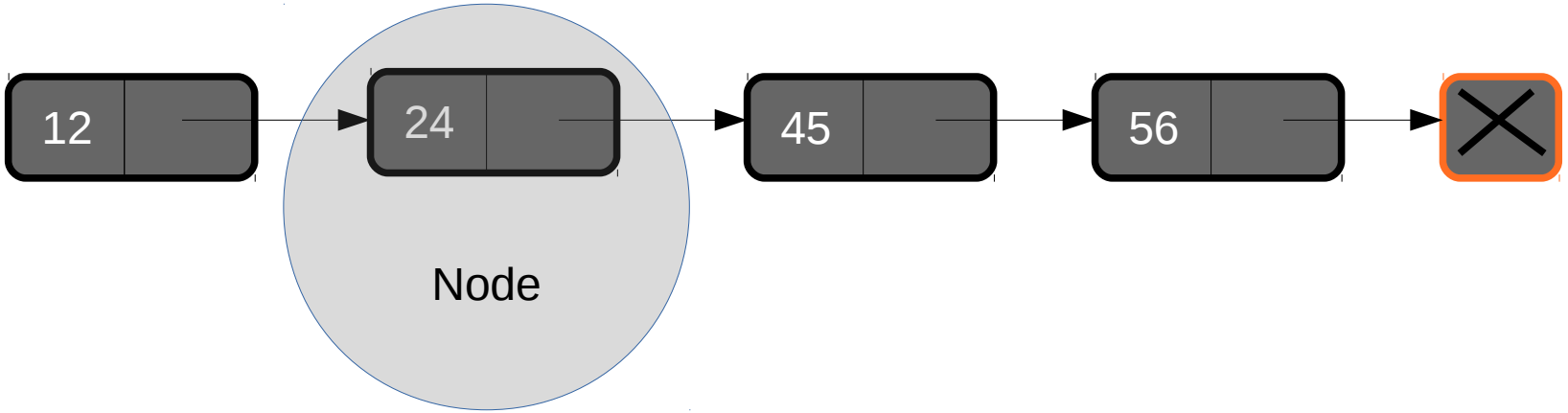
Stages:Program Design

- Identify the data structures.
- Operations: Algorithms
- Efficiency(Complexity)
- Implementation
  - What goes into header file?
  - What goes into C program?
  - What are libraries? Why do we need them?
  - How to create libraries.

Chapter 2: Linked List

Abstract

- A collection of items accessible one after another beginning at the head and ending at the tail is called a list.
- A **linked list** is a data structure consisting of a group of nodes which together represent a sequence.
- Under the simplest form, each node is composed of a data and a reference (in other words, a link) to the next node in the sequence.



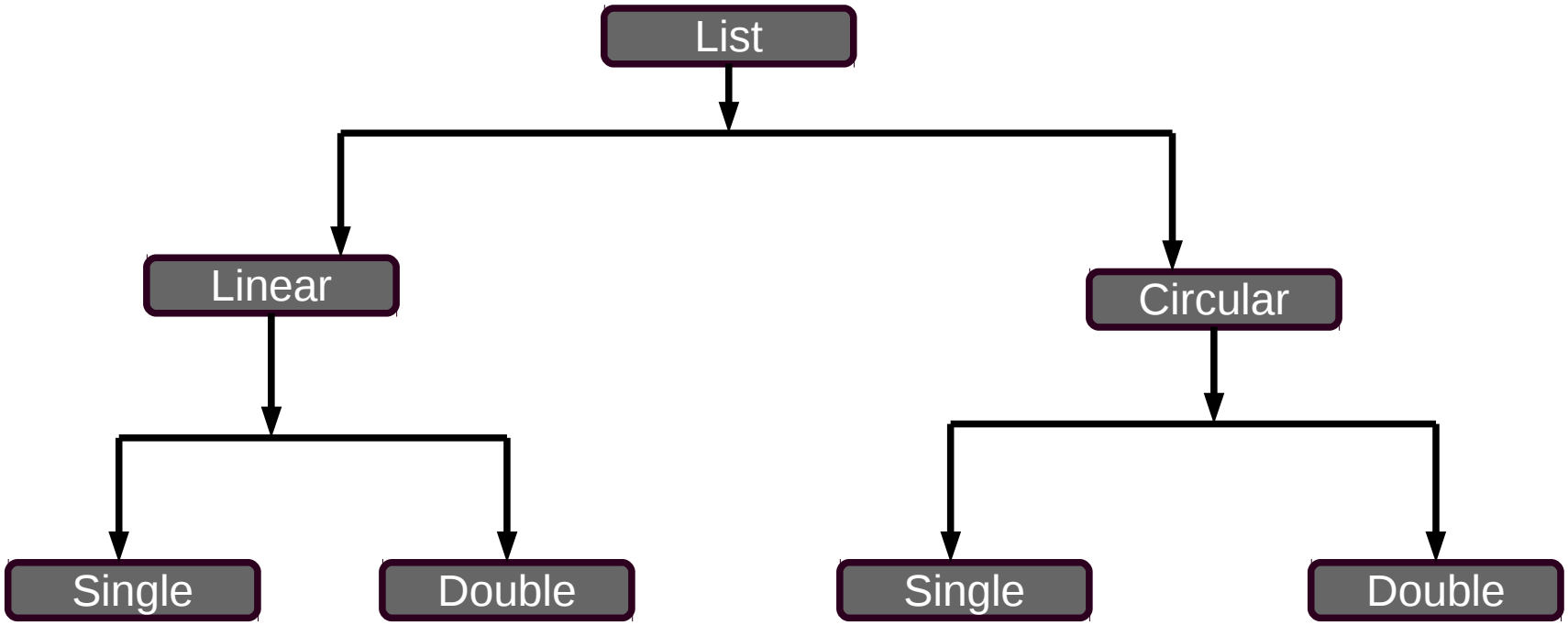
ΣMERTXE

Why : Linked List

Linked List Vs Arrays

- Elements can be inserted into linked lists indefinitely, while an array will eventually either fill up or need to be resized.
- Further memory savings can be achieved.
- Simple example of a persistent data structure.
- On the other hand, arrays allow random access, while linked lists allow only sequential access to elements.
- Another disadvantage of linked lists is the extra storage needed for references, which often makes them impractical for lists of small data items such as characters or boolean values.

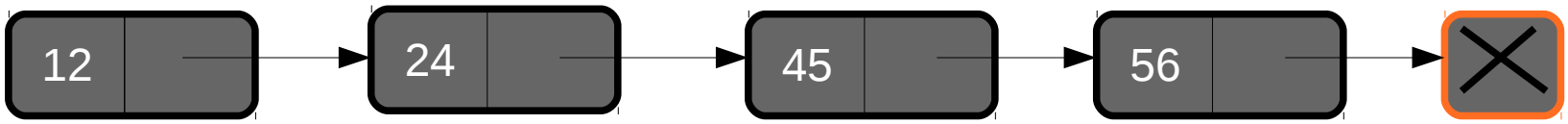
Linked List : Types



Singly Linked List

- The simplest kind of linked list is a singly-linked list (or slist for short), which has one link per node.
- This link points to the next node in the list, or to a null value or empty list if it is the final node.

Example

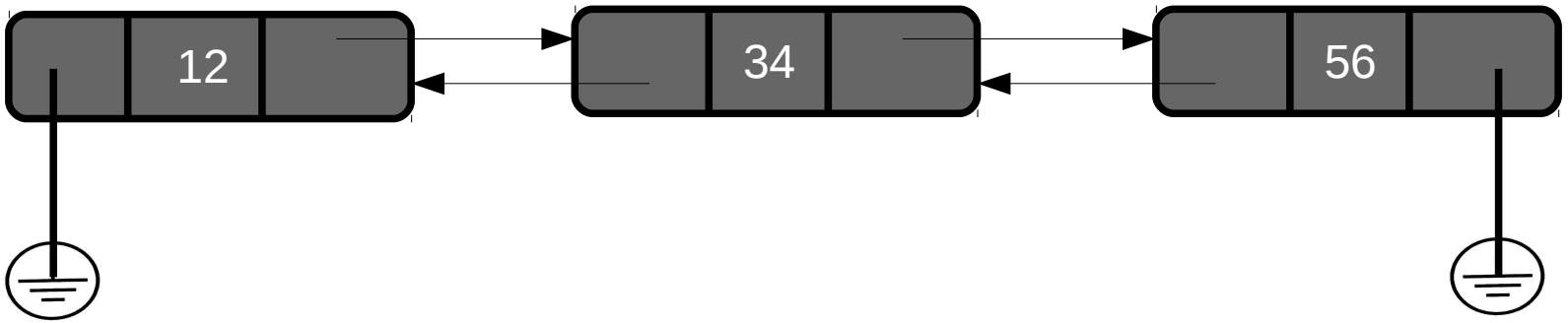


EMERTXE

Doubly Linked List

- A variant of a linked list in which each item has a link to the previous item as well as the next.
- This allows easily accessing list items backward as well as forward and deleting any item in constant time, also known as two-way linked list, symmetrically linked list.

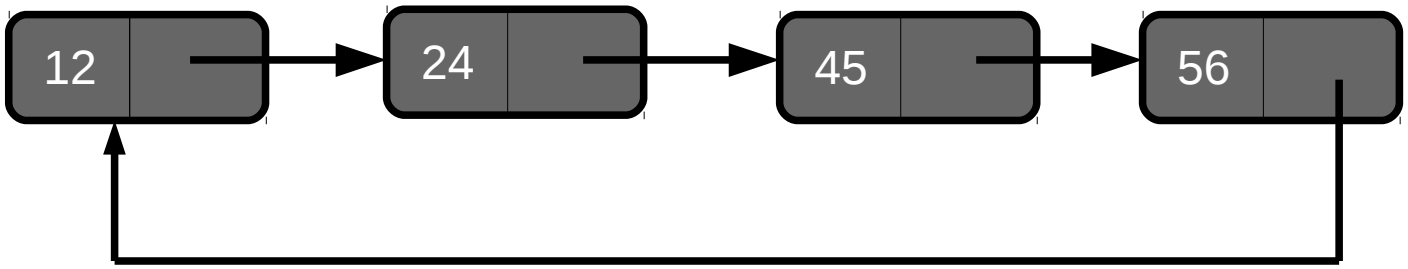
Example



Singly Circular Linked List

- Similar to an ordinary singly-linked list, except that the next link of the last node points back to the first node.

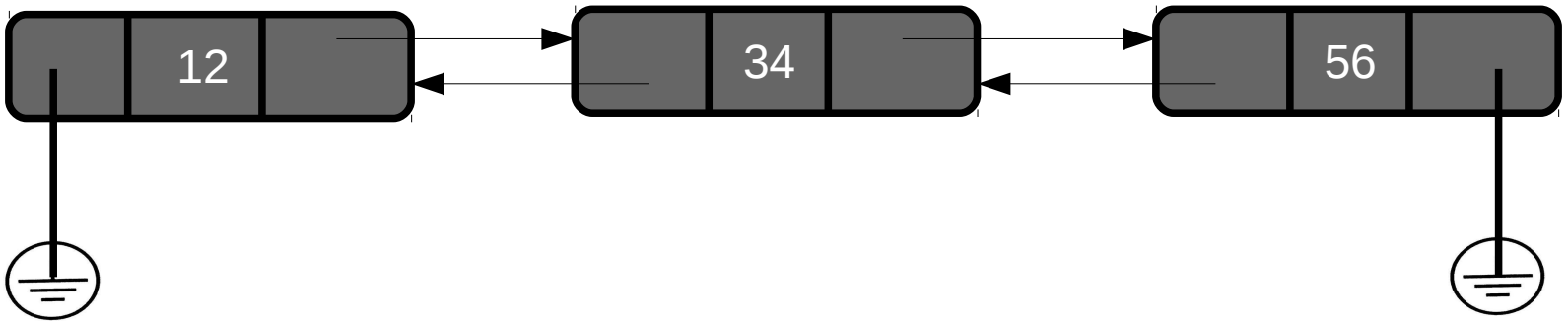
Example



Doubly Linked List

- A variant of a linked list in which each item has a link to the previous item as well as the next.
- This allows easily accessing list items backward as well as forward and deleting any item in constant time, also known as two-way linked list, symmetrically linked list.

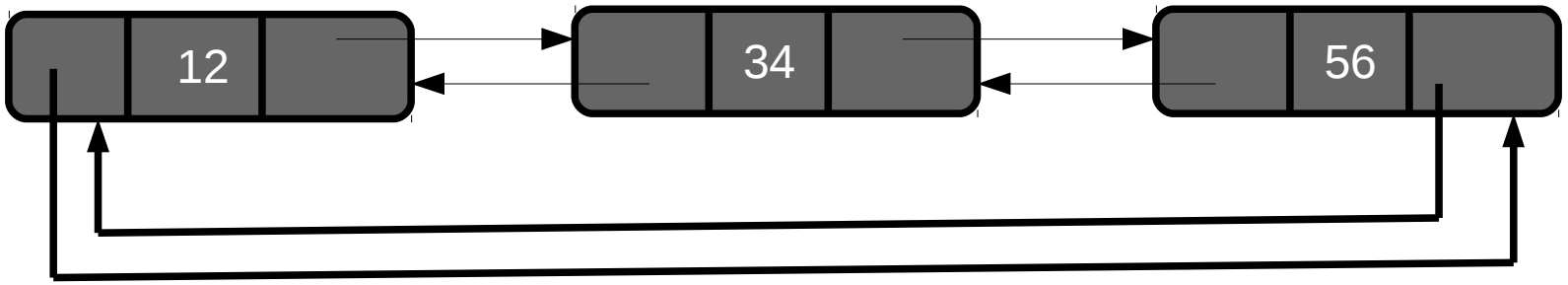
Example



Doubly Circular Linked List

- Similar to a doubly-linked list, except that the previous link of the first node points to the last node and the next link of the last node points to the first node.

Example



TradeOffs

Double Linked List Vs Single Linked List

Single Linked List	*Double Linked List*
Less space per node	More space per node
Elementary operations Less expensive	Elementary operations more expensive
Bit difficult to manipulate	Easier to manipulate

Circular Linked List Vs Linear Linked List

- Allows quick access to the first and last records through a single pointer (the address of the last element).
- Their main disadvantage is the complexity of iteration, which has subtle special cases.

Operations : Linked List

1. Creation
2. Insertion
3. Deletion
4. Searching
5. Sorting
6. Listing
7. Destroying etc...

Applications

Used in the implementation of other Data Structures

1. Stacks

2. Queues

Chapter 3: Stack

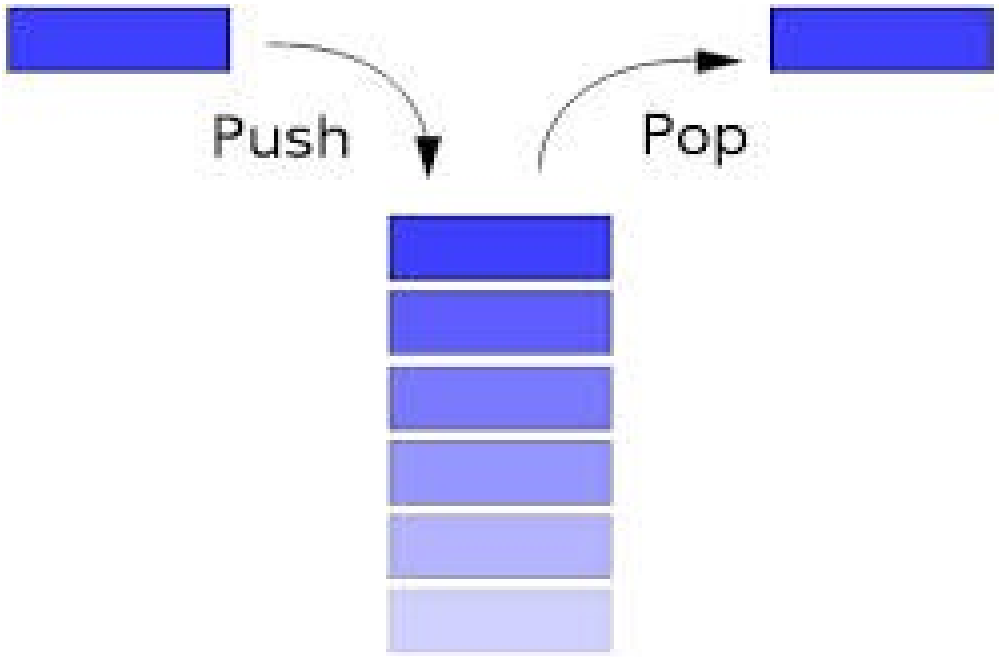
Abstract

- Stack is a collection of items in which only the most recently added item may be removed.
- The latest added item is at the top.
- Basic operations are **push** and **pop**.
- Also known as **last-in, first-out** or **LIFO**.

EMERTXE

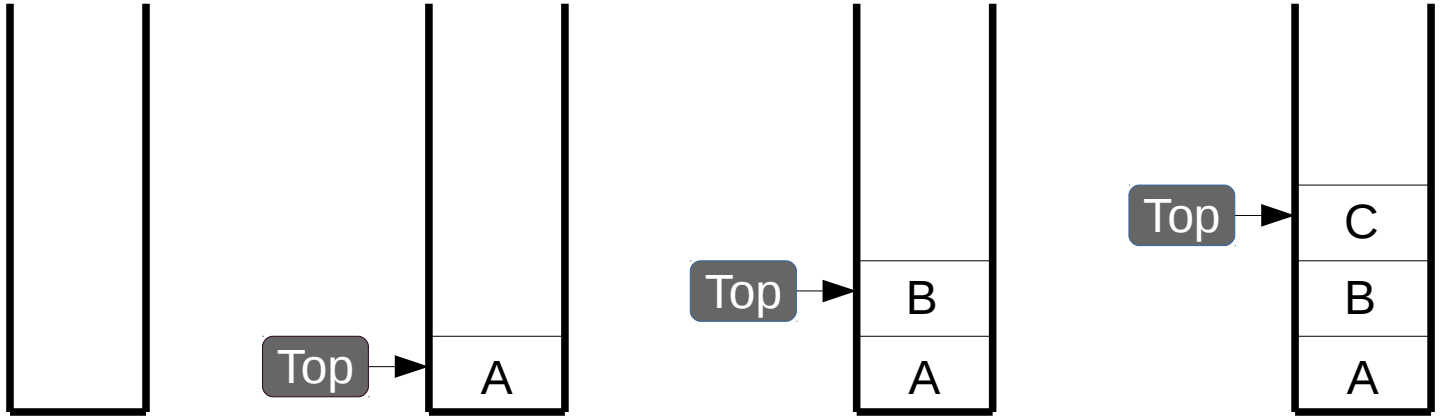
Abstract ...

- Simply stack is a memory in which value are stored and retrieved in **last in first out** manner by using operations called **push** and **pop**.



Stack : Operations

Push Operations

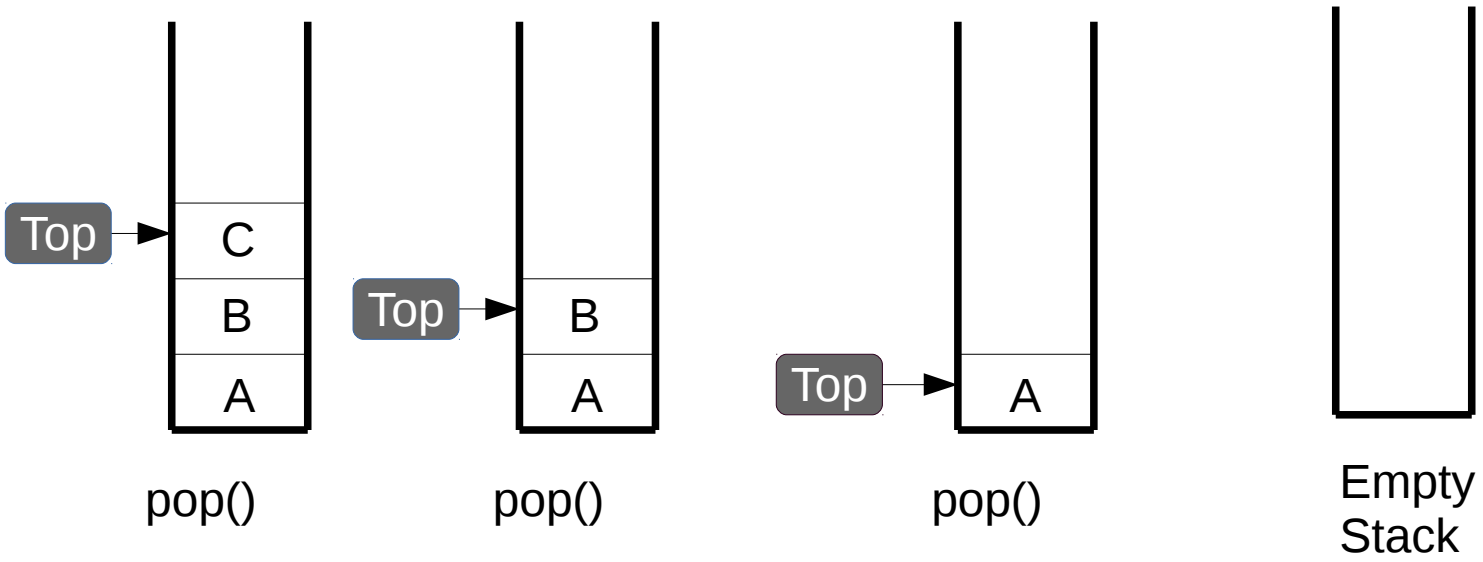


OVERFLOW STATE

If the stack is full and does not contain enough space to accept the given item

Stack : Operations

Pop Operations

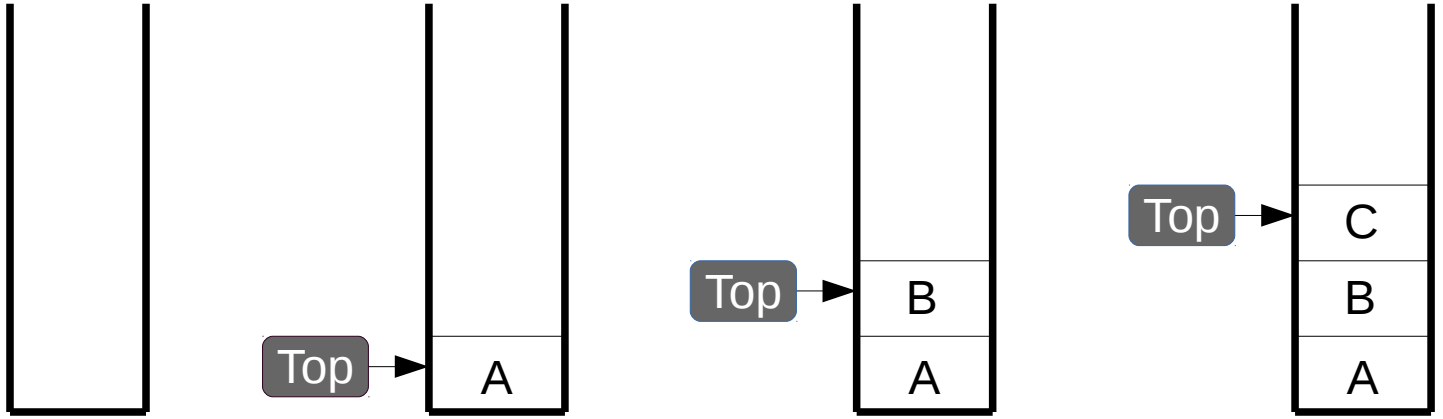


UNDERFLOW STATE

If the stack is empty and performing pop() operation results in underflow.

Stack : Operations

Push Operations

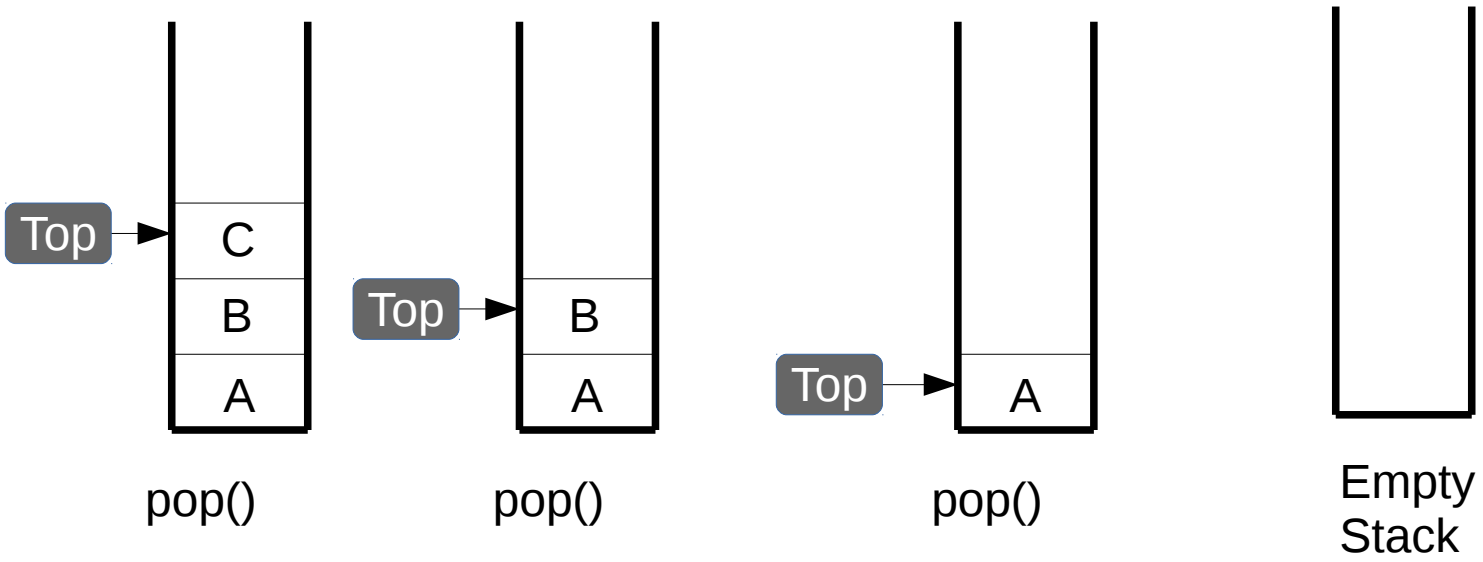


OVERFLOW STATE

If the stack is full and does not contain enough space to accept the given item

Stack : Operations

Pop Operations



UNDERFLOW STATE

If the stack is empty and performing pop() operation results in underflow.

Stack : Operations

1. Create the stack
2. Add to the stack
3. Delete from the stack
4. Print the stack
5. Destroy the stack

Applications

1. Decimal to Binary conversion

2. Conversion of expressions
   - Infix – Postfix
   - Infix – Prefix

3. Evaluation of expressions
   - Infix expression evaluation.
   - Prefix expression evaluation.
   - Postfix expression evaluation.

4. Function calls in C

Chapter 4: Searching Techniques

Search Algorithms

"A search algorithm is a method of locating a specific item of information in a larger collection of data. "

EMERTXE

Linear Search

- This is a very simple algorithm.
- It uses a loop to sequentially step through an array, starting with the first element.
- It compares each element with the value being searched for and stops when that value is found or the end of the array is reached.

Efficiency: Linear Search

The Advantage is its simplicity.

- It is easy to understand
- Easy to implement
- Does not require the array to be in order

The Disadvantage is its inefficiency

- If there are 20,000 items in the array and what you are looking for is in the 19,999th element, you need to search through the entire list.

Time Complexities

Cases	Big - O
Best	O(1)
Average	O(N / 2)
Worst	O(N)

Binary Search OR Half-Interval Search

- The binary search is much more efficient than the linear search.
- It requires the list to be in order
- The algorithm starts searching with the middle element.
  - If the item is less than the middle element, it starts over searching the first half of the list.
  - If the item is greater than the middle element, the search starts over starting with the middle element in the second half of the list.
  - It then continues halving the list until the item is found.

Binary Search Algorithm

Find the index of the element x in the **ordered** list $\{a_1 \leq a_2 \leq a_3 \leq \cdots \leq a_n\}$.

1. Split the list into two sublists $a_1, a_2, \ldots, a_{\lfloor n/2 \rfloor}$ and $a_{\lfloor n/2 \rfloor + 1}, \ldots, a_n$.
2. If $x < a_{\lfloor n/2 \rfloor}$, then x is in the first sublist. Otherwise it is in $a_{\lfloor n/2 \rfloor + 1}, \ldots, a_n$.
3. Split the new list into two lists, and proceed as above.
4. Stop when you find that x is in a list of length 1.

Efficiency: Binary Search

Advantage

- It uses “Divide & Conquer” technique.
- Binary search uses the result of each comparison to eliminate half of the list from further searching.
- Binary search reveals whether the target is before or after the current position in the list, and that information can be used to narrow the search.
- Binary search is significantly better than linear search for large lists of data
- Binary search maintains a contiguous subsequence of the starting sequence where the target value is surely located.

Efficiency: Binary Search

Disdvantage

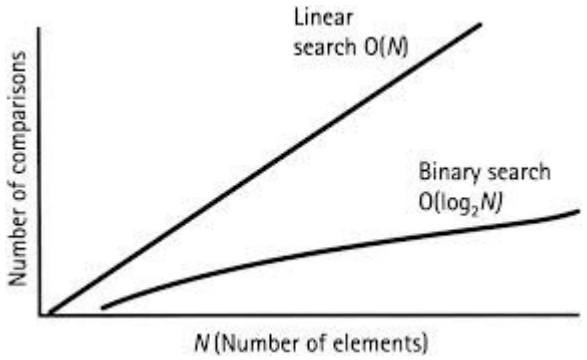
- Binary search can interact poorly with the memory hierarchy (i.e. caching), because of its random-access nature.
- For in-memory searching, if the interval to be searching is small, a linear search may have superior performance simply because it exhibits better locality of reference.
- Binary search algorithm employs recursive approach and this approach requires more stack space.

Time Complexities

Cases	Big - O
Best	O (1)
Average	O (log n)
Worst	O (log n)

Linear Vs Binary

- The benefit of binary search over linear search becomes significant for lists over about 100 elements.
- For smaller lists linear search may be faster because of the speed of the simple increment compared with the divisions needed in binary search.
- The general moral is that for large lists binary search is very much faster than linear search, but is not worth while for small lists.
- Note that binary search is not appropriate for linked list structures (no random access for the middle term).



Chapter 4: Sorting Techniques

Abstract

"A sorting algorithm is an algorithm that puts elements of a list in a certain order. "

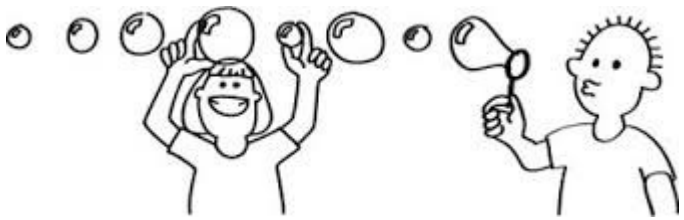
EMERTXE

Sorting Techniques

- Bubble sort
- Insertion sort
- Select sort
- Quick sort
- Merge sort
- Bucket sort
- Radix sort

Bubble Sort

- **Bubble sort**, sometimes referred to as **sinking sort**, is a simple sorting algorithm.
- Works by repeatedly stepping through the list to be sorted, comparing each pair of adjacent items and swapping them if they are in the wrong order.
- The pass through the list is repeated until no swaps are needed, which indicates that the list is sorted.
- The algorithm gets its name from the way smaller elements "bubble" to the top of the list.



Algorithm

```
procedure bubbleSort( A : list of sortable items ) defined as:
 do
       swapped := false
       for each i in 0 to length( A ) - 1 do:
       if A[ i ] > A[ i + 1 ] then
             swap( A[ i ], A[ i + 1 ] )
             swapped := true
       end if
       end for
 while swapped
 end procedure
```

Example

Let us take the array of numbers "5 1 4 2 8"

First Pass:

(5 1 4 2 8) -> (1 5 4 2 8),
//Here, algorithm compares the first two elements, and swaps since 5 > 1.

(1 5 4 2 8) -> (1 4 5 2 8), Swap since 5 > 4

(1 4 5 2 8) -> (1 4 2 5 8), Swap since 5 > 2

(1 4 2 5 8) -> (1 4 2 5 8),
//Now, since these elements are already in order (8 > 5), algorithm does not swap them.

Example

Second Pass:

(1 4 2 5 8) -> (1 4 2 5 8)

(1 4 2 5 8) -> (1 2 4 5 8), Swap since 4 > 2

(1 2 4 5 8) -> (1 2 4 5 8)

(1 2 4 5 8) -> (1 2 4 5 8)

Now, the array is already sorted, but our algorithm does not know if it is completed. The algorithm needs one whole pass without any swap to know it is sorted.

Example

Third Pass:

(1 2 4 5 8) -> (1 2 4 5 8)

(1 2 4 5 8) -> (1 2 4 5 8)

(1 2 4 5 8) -> (1 2 4 5 8)

(1 2 4 5 8) -> (1 2 4 5 8)

Performance Analysis

- Bubble sort has worst-case and average complexity both O(n2), where n is the number of items being sorted.
- There exist many sorting algorithms with substantially better worst-case or average complexity of O(n log n)
- Even other O(n2) sorting algorithms, such as insertion sort, tend to have better performance than bubble sort.
- Therefore, bubble sort is not a practical sorting algorithm when n is large.

Performance Analysis

Advantage

- The only significant advantage that bubble sort has over most other implementations, even quicksort, but not insertion sort, is that the ability to detect that the list is sorted is efficiently built into the algorithm.
- Although bubble sort is one of the simplest sorting algorithms to understand and implement, its O(n2) complexity means that its efficiency decreases dramatically on lists of more than a small number of elements.
- Due to its simplicity, bubble sort is often used to introduce the concept of an algorithm, or a sorting algorithm, to introductory computer science students

Time Complexities

Cases	Big - O
Best	O (n)
Average	O (n^2)
Worst	O (n^2)

Insertion Sort

"Insertion sort is a simple sorting algorithm that builds the final sorted array (or list) one item at a time"

Algorithm

```
insertionSort(array A)
for i = 1 to length[A]-1 do
begin
        value = A[i]
        j = i-1
        while j >= 0 and A[j] > value do
        begin
                swap( A[j + 1], A[j] )
                j = j-1
        end
        A[j+1] = value
end
```

EMERTXE

Example

Consider the sequence, {3, 7, 4, 9, 5, 2, 6, 1}

3 7 4 9 5 2 6 1

3 7 4 9 5 2 6 1

3 7 4 9 5 2 6 1

3 4 7 9 5 2 6 1

3 4 7 9 5 2 6 1

3 4 5 7 9 2 6 1

2 3 4 5 7 9 6 1

2 3 4 5 6 7 9 1

1 2 3 4 5 6 7 9

Performance Analysis

Advantages

- Simple implementation
- Efficient for (quite) small data sets
- Adaptive (i.e., efficient) for data sets that are already substantially sorted: the time complexity is O(n + d), where d is the number of inversions
- More efficient in practice than most other simple quadratic (i.e., O(n2)) algorithms such as selection sort or bubble sort; the best case (nearly sorted input) is O(n)
- Stable; i.e., does not change the relative order of elements with equal keys
- In-place; i.e., only requires a constant amount O(1) of additional memory space
- Online; i.e., can sort a list as it receives it

Performance Analysis

Dis-Advantages

- Insertion sort typically makes fewer comparisons than selection sort, it requires more writes because the inner loop can require shifting large sections of the sorted portion of the array.

- In general, insertion sort will write to the array O(n2) times, whereas selection sort will write only O(n) times.

- For this reason selection sort may be preferable in cases where writing to memory is significantly more expensive than reading, such as with EEPROM or flash memory.

Time Complexities

Cases	Big - O
Best	O (n)
Average	O (n^2)
Worst	O (n^2)

Selection Sort

"selection sort is a sorting algorithm, specifically an in-place comparison sort"

The idea of selection sort is rather simple:

we repeatedly find the next largest (or smallest) element in the array and move it to its final position in the sorted array

Algorithm

for i = 1:n,

k = i

for j = i+1:n, if a[j] < a[k], k = j

→ invariant: a[k] smallest of a[i..n]

swap a[i,k]

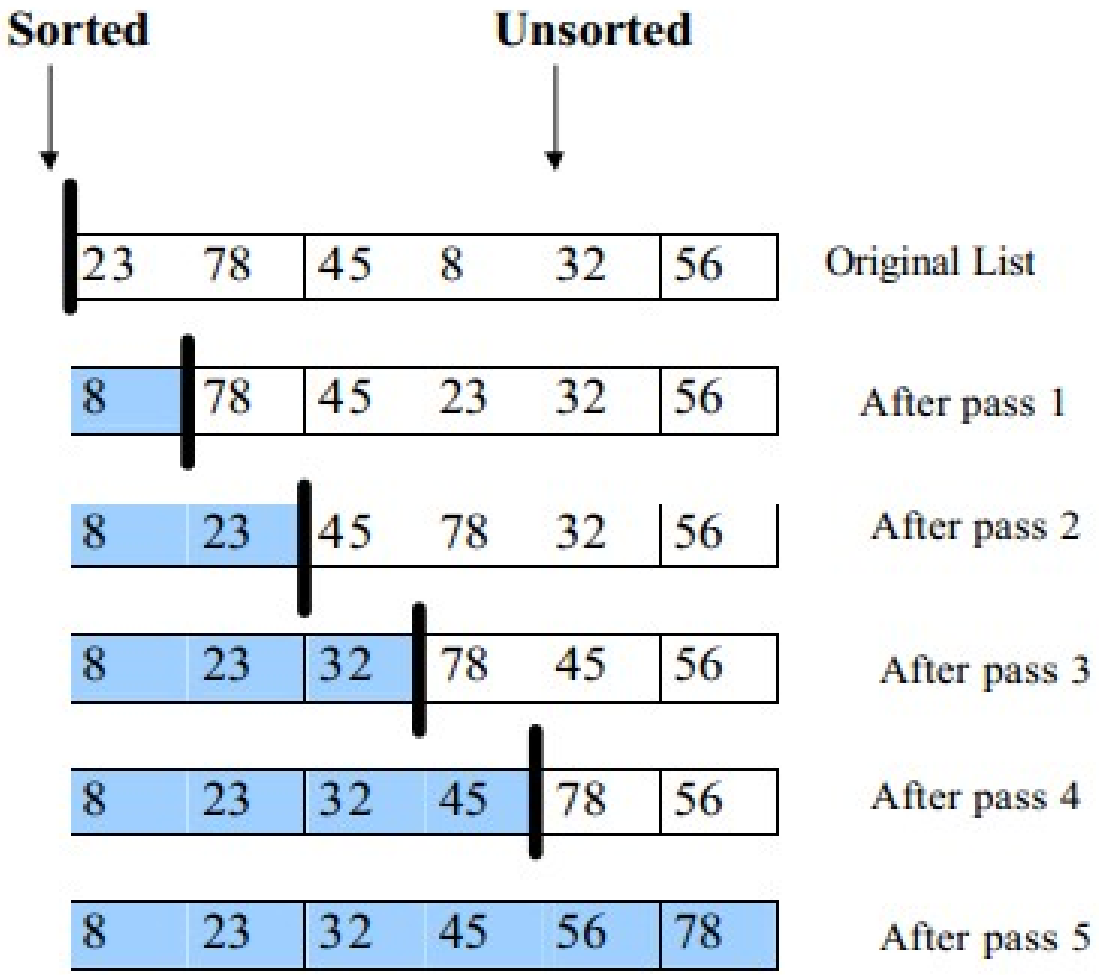
→ invariant: a[1..i] in final position

end

Example

- The list is divided into two sublists, sorted and unsorted, which are divided by an imaginary wall.
- We find the smallest element from the unsorted sublist and swap it with the element at the beginning of the unsorted data.
- After each selection and swapping, the imaginary wall between the two sublists move one element ahead, increasing the number of sorted elements and decreasing the number of unsorted ones.
- Each time we move one element from the unsorted sublist to the sorted sublist, we say that we have completed a sort pass.
- A list of n elements requires n-1 passes to completely rearrange the data.

Example


Sorted
Unsorted
23 78 45 8 32 56
Original List
8 78 45 23 32 56
After pass 1
8 23 45 78 32 56
After pass 2
8 23 32 78 45 56
After pass 3
8 23 32 45 78 56
After pass 4
8 23 32 45 56 78
After pass 5

Performance Analysis

- Selecting the lowest element requires scanning all n elements (this takes n − 1 comparisons) and then swapping it into the first position

Time Complexities

Cases	Big - O
Best	$O(n^2)$
Average	$O(n^2)$
Worst	$O(n^2)$

Quick Sort

"As the name implies, it is quick, and it is the algorithm generally preferred for sorting."

Basic Idea **[Divide & Conquer Technique]**

1. Pick an element, say P (the pivot)

2. Re-arrange the elements into 3 sub-blocks,
   - those less than or equal to P (left block)
   - P(the only element in the middle block)
   - Those greater than P(right block)

3. Repeat the process recursively for the left & right sub blocks.

Algorithm

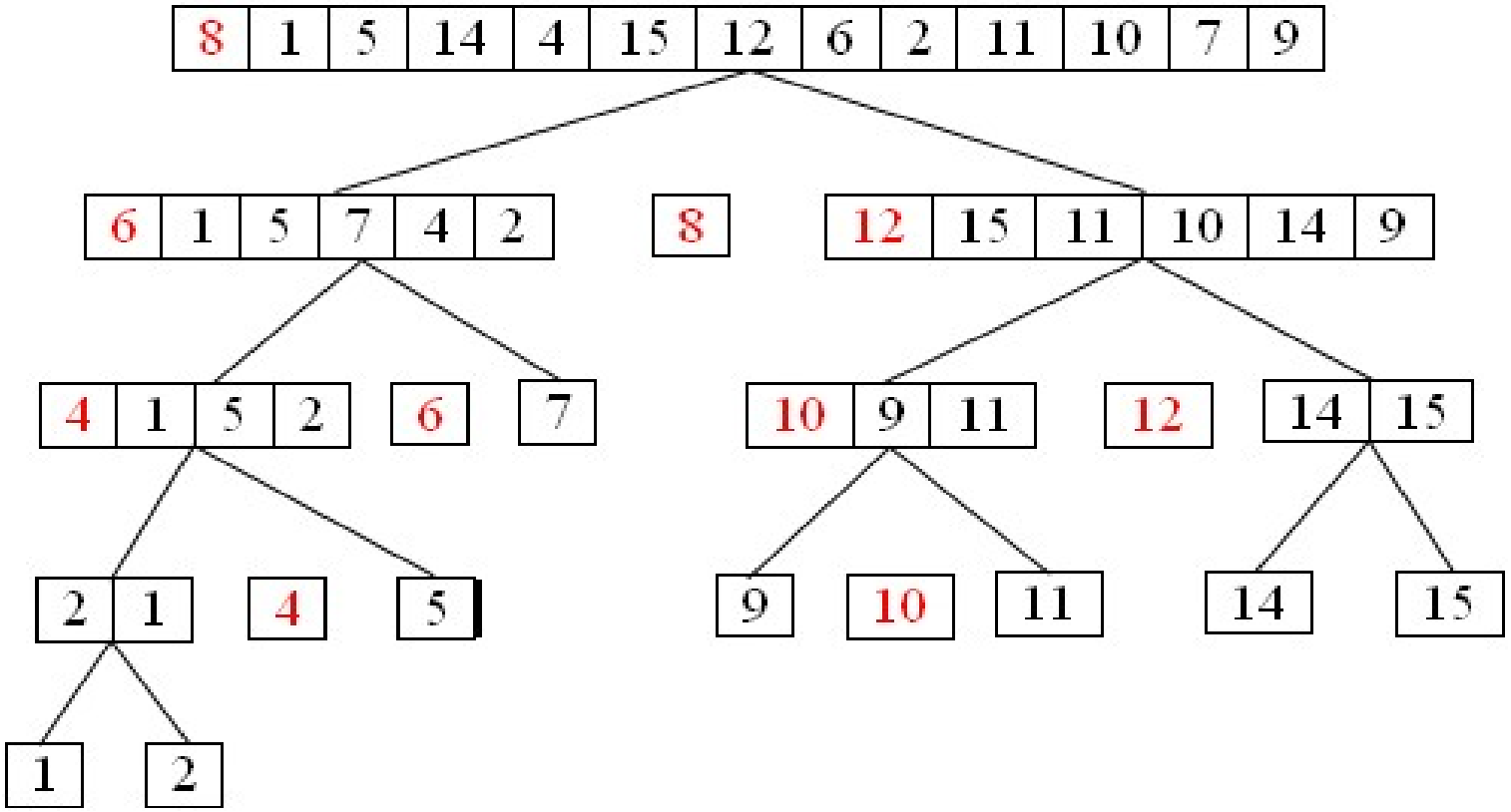
```
function partition(array, left, right, pivotIndex)
    pivotValue := array[pivotIndex]
    swap array[pivotIndex] and array[right] // Move pivot to end
    storeIndex := left
    for i  from  left to right ? 1
        if array[i] ? pivotValue
            swap array[i] and array[storeIndex]
            storeIndex := storeIndex + 1
    swap array[storeIndex] and array[right] // Move pivot to its final place
    return storeIndex
```

Algorithm

```
procedure quicksort(array, left, right)
    if right > left
        select a pivot index (e.g. pivotIndex := left)
        pivotNewIndex := partition(array, left, right, pivotIndex)
        quicksort(array, left, pivotNewIndex - 1)
        quicksort(array, pivotNewIndex + 1, right)
```

ΣMERTXE

Example



1 2 4 5 6 7 8 9 10 11 12 14 15

Advantages

Plus Points

- It is in-place since it uses only a small auxiliary stack.
- It requires only n log(n) time to sort n items.
- It has an extremely short inner loop
- This algorithm has been subjected to a thorough mathematical analysis, a very precise statement can be made about performance issues.

Disadvantages

Minus Points

- It is recursive. Especially if recursion is not available, the implementation is extremely complicated.
- It requires quadratic (i.e., n2) time in the worst-case.
- It is fragile i.e., a simple mistake in the implementation can go unnoticed and cause it to perform badly.

Time Complexities

Cases	Big - O
Best	O (n log n)
Average	O (n log n)
Worst	O (n^2)

Merge Sort

"MergeSort is a Divide and Conquer algorithm. It divides input array in two halves, calls itself for the two halves and then merges the two sorted halves."

Basic Idea

Divide & Conquer Technique

Algorithm

Conceptually, a merge sort works as follows:

- Divide the unsorted list into n sublists, each containing 1 element (a list of 1 element is considered sorted).
- Repeatedly merge sublists to produce new sorted sublists until there is only 1 sublist remaining. This will be the sorted list.

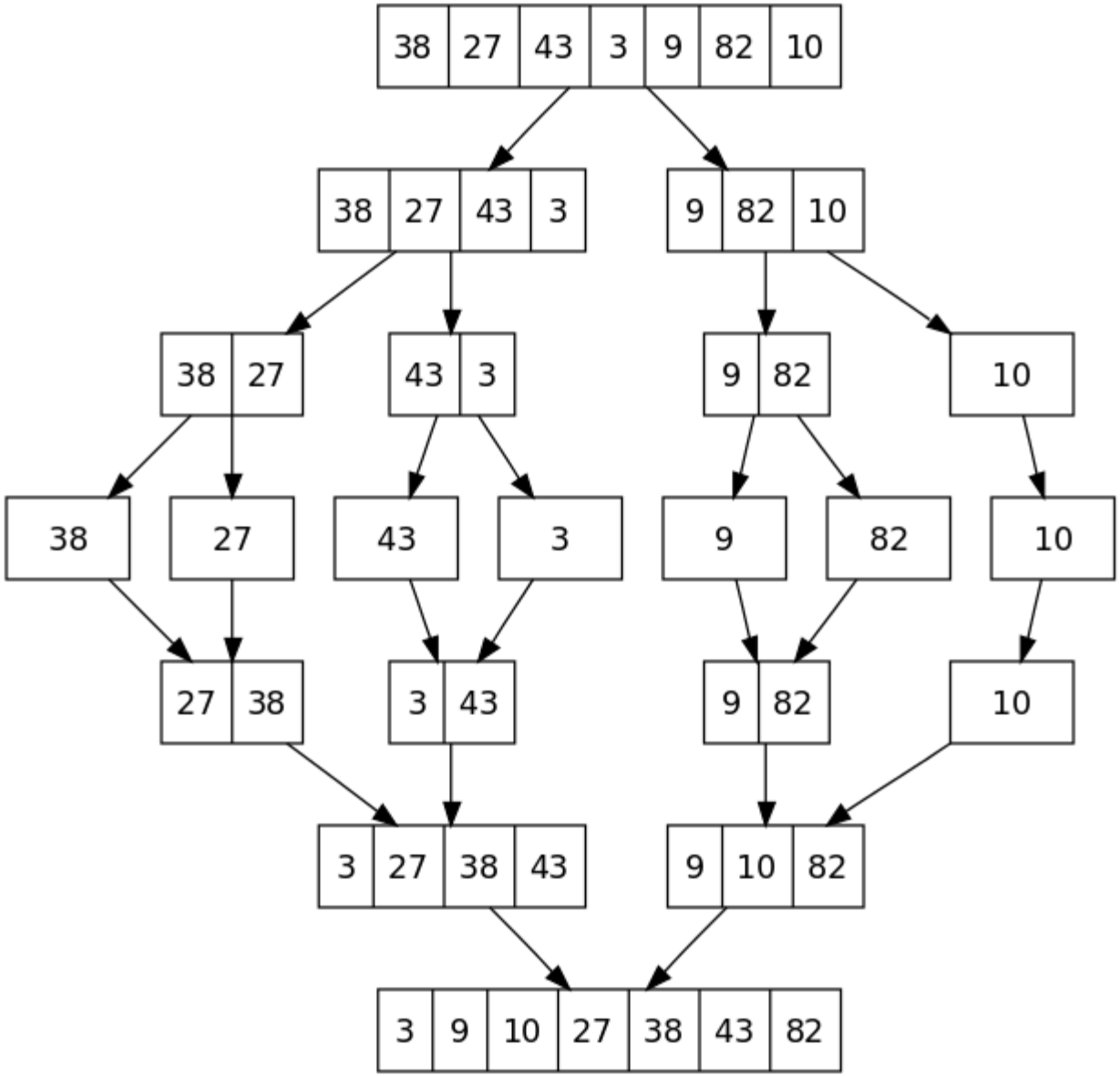
Algorithm

```
function mergesort(m)
    var list left, right, result
    if length(m) ? 1
       return m
    var middle = length(m) / 2
    for each x in m up to middle
        add x to left
    for each x in m after middle
        add x to right
    left = mergesort(left)
    right = mergesort(right)
    result = merge(left, right)
    return result
```

Algorithm

```
function merge(left,right)
        var list result
        while length(left) > 0 and length(right) > 0
            if first(left) ? first(right)
                append first(left) to result
                left = rest(left)
            else
                append first(right) to result
                right = rest(right)
        end while
        if length(left) > 0
            append rest(left) to result
        if length(right) > 0
            append rest(right) to result
        return result
```

Example



EMERTXE

Applications

1) Merge Sort is useful for sorting linked lists in O(nLogn) time. Other nlogn algorithms like Heap Sort, Quick Sort (average case nLogn) cannot be applied to linked lists.

2) Inversion Count Problem

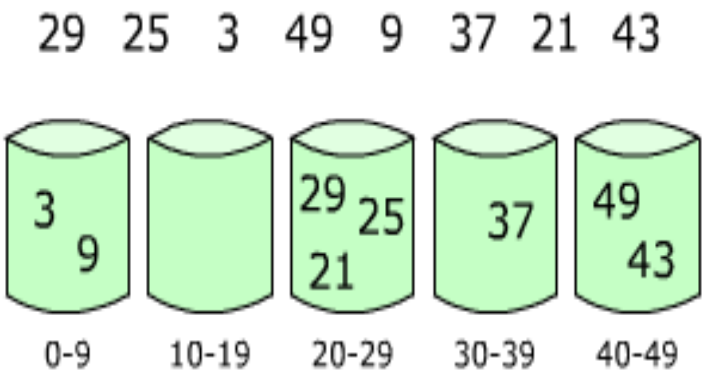
3) Used in External Sorting

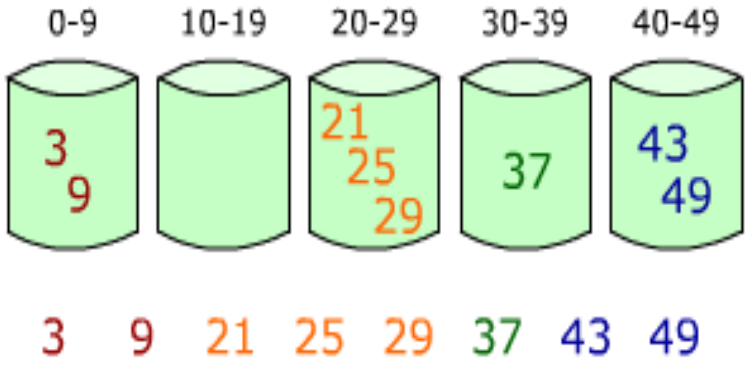
Time Complexities

Cases	Big - O
Best	O (n log n)
Average	O (n log n)
Worst	O (n log n)

Bucket Sort

"Bucket sort, or bin sort, is a sorting algorithm that works by partitioning an array into a number of buckets"





ΣMERTXE

Bucket Sort : Idea

Idea:

- suppose the values are in the range 0..m-1; start with m empty *buckets* numbered 0 to m-1, scan the list and place element s[i] in bucket s[i], and then output the buckets in order.

- Will need an array of buckets, and the values in the list to be sorted will be the indexes to the buckets

- No comparisons will be necessary

Algorithm

Bucket sort works as follows:

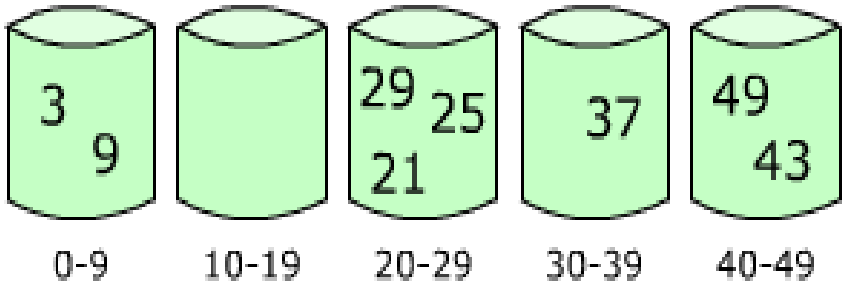
- Set up an array of initially empty "buckets".
- Scatter: Go over the original array, putting each object in its bucket.
- Sort each non-empty bucket.
- Gather: Visit the buckets in order and put all elements back into the original array.

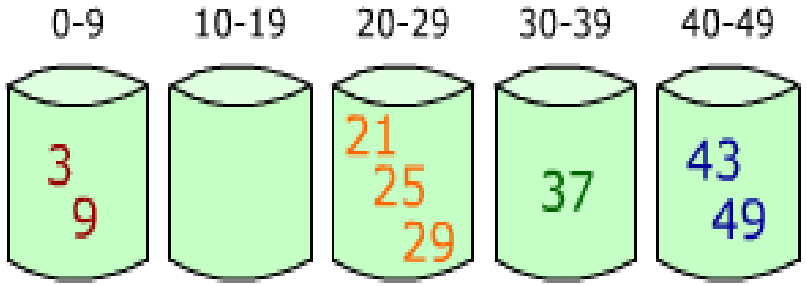
Algorithm: Pseudocode

```
function bucketSort(array, n) is
  buckets ← new array of n empty lists
  for i = 0 to (length(array)-1) do
    insert array[i] into buckets[msbits(array[i], k)]
  for i = 0 to n - 1 do
    nextSort(buckets[i]);
  return the concatenation of buckets[0], ...., buckets[n-1]
```

Example

29 25 3 49 9 37 21 43





3 9 21 25 29 37 43 49

Time Complexities

Cases	Big - O
Average	O (n + k)
Worst	O (n^2)

Radix Sort

"Radix sort is a non-comparative integer sorting algorithm that sorts data with integer keys by grouping keys by the individual digits which share the same significant position and value"

Algorithm

RADIX_SORT (A, d)

```
for i ← 1 to d do
    use a stable sort to sort A on digit i
    // counting sort will do the job
```

Example

Following example shows how Radix sort operates on seven 3-digits number.

Input	1st pass	2nd pass	3rd pass
329	720	720	329
457	355	329	355
657	436	436	436
839	457	839	457
436	657	355	657
720	329	457	720
355	839	657	839

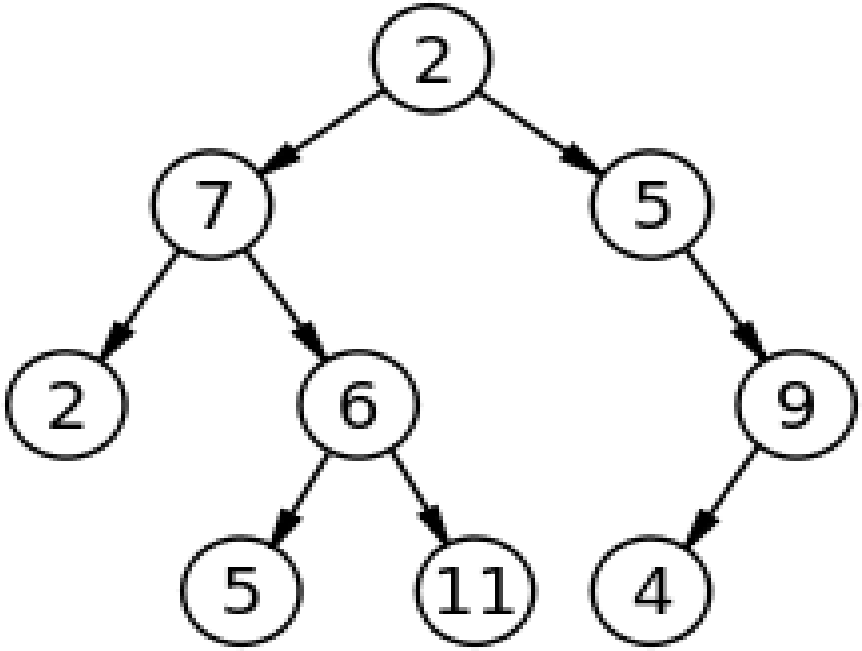
Time Complexities

Cases	Big - O
Worst case time complexity	O (n*k)
Worst Case space complexity	O (n + k)

Chapter 5: Trees

Abstract

“A tree is a widely used abstract data type (ADT) or data structure implementing this ADT that simulates a hierarchical tree structure, with a root value and subtrees of children, represented as a set of linked nodes. “



ΣMERTXE

Abstract

- A data structure accessed beginning at the root node.
- Each node is either a leaf or an internal node.
- An internal node has one or more child nodes and is called the parent of its child nodes.
- All children of the same node are siblings.
- Contrary to a physical tree, the root is usually depicted at the top of the structure, and the leaves are depicted at the bottom.

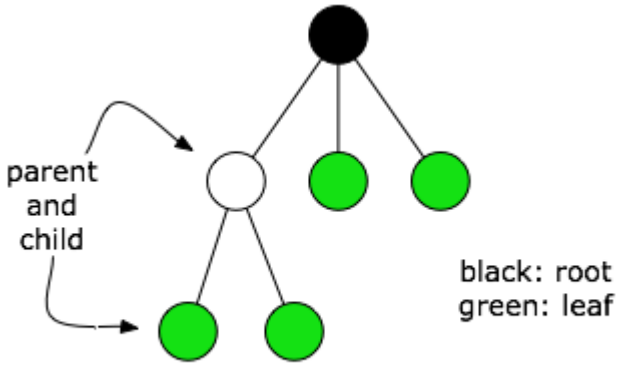


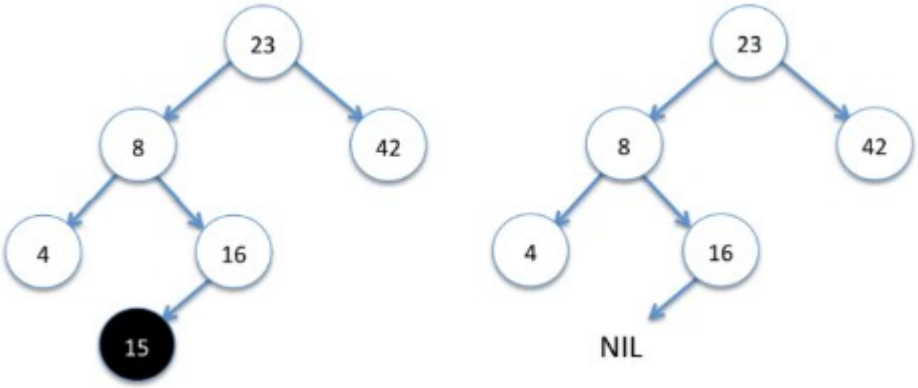
Figure: tree data structure

Operations:
Binary Tree

- Create new node
- Insert into the tree
- Delete from the tree
  - Deleting a leaf
  - Deleting the node with one child
  - Deleting the node with two children

Operations:
Binary Tree

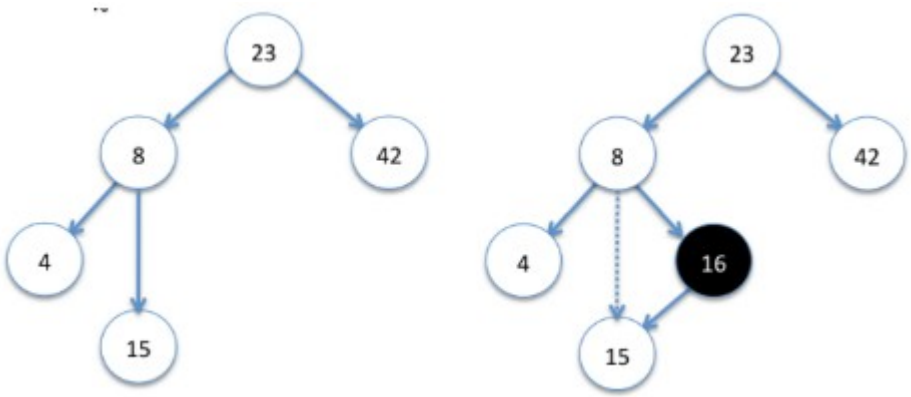
- Delete from the tree
  - Deleting a leaf



EMERTXE

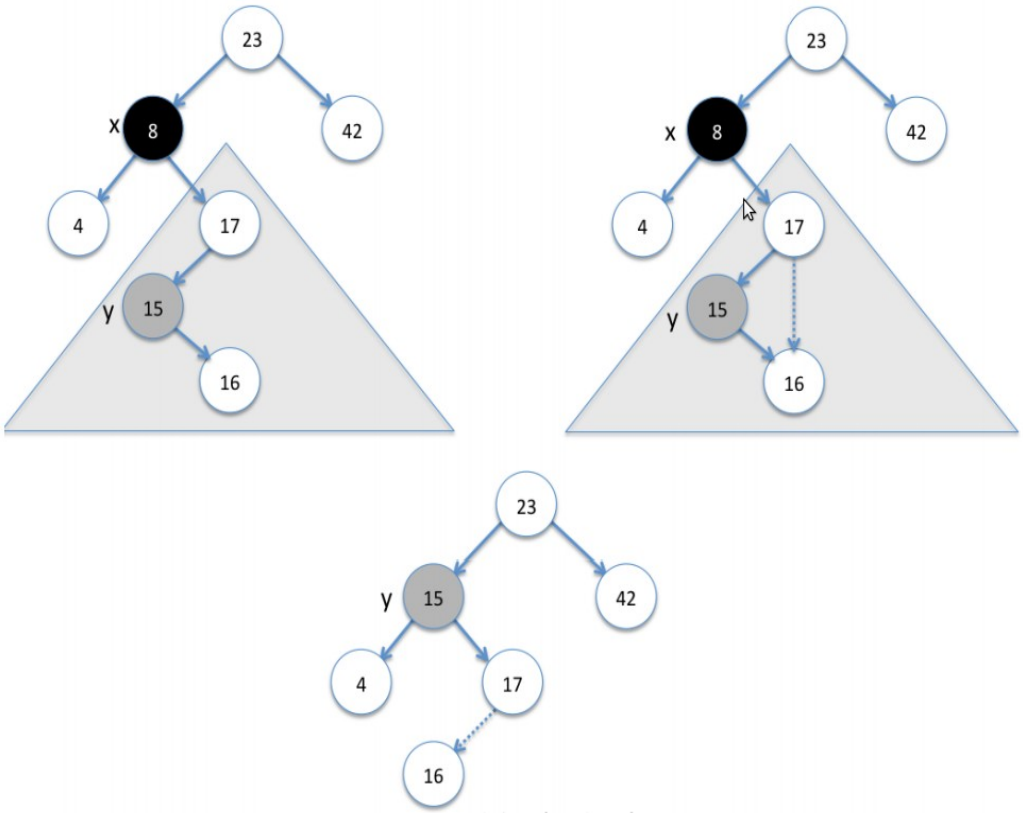
Operations:
Binary Tree

- Delete from the tree
  - Deleting the node with one child



Operations:
Binary Tree

- Delete from the tree
  - Deleting the node with two children



Traversal: Binary Tree

Inorder

```
if the tree is not empty
        traverse the left subtree
        visit the root
        traverse the right subtree
```

preorder

```
if the tree is not empty
        visit the root
        traverse the left subtree
        traverse the right subtree
```

postorder

```
if the tree is not empty
        traverse the left subtree
        traverse the right subtree
        visit the root
```

Applications:
Binary Tree

- Storing a set of names, and being able to lookup based on a prefix of the name. (Used in internet routers.)
- Storing a path in a graph, and being able to reverse any subsection of the path in O(log n) time. (Useful in travelling salesman problems).
- Use in Heap sorting.

Heap Sort:

- A sort algorithm that builds a heap, then repeatedly extracts the maximum item.
- Run time is O(n log n).
- A kind of in-place sort.

Chapter 6: Hashing

Abstract

“Hashing is a method to store data in an array so that storing, searching,inserting and deleting data is fast (in theory it’s O(1)). For this every record needs an unique key.“

Basic Idea

Not to search for the correct position of a record with comparisons but to compute the position within the array.

The function that returns the position is called the '***hash function***' and the array is called a '***hash table***'.

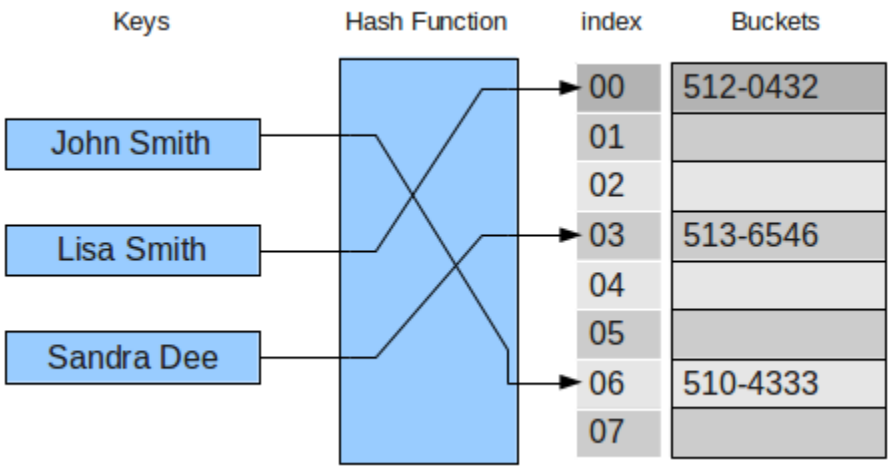
ΣMERTXE

Hash Function

- A function that maps keys to integers, usually to get an even distribution on a smaller set of values.
- A hash table or hash map is a data structure that uses a hash function to map identifying values, known as keys (e.g., a person's name), to their associated values (e.g., their telephone number).
- Thus, a hash table implements an associative array.
- The hash function is used to transform the key into the index (the hash) of an array element (the slot or bucket) where the corresponding value is to be sought.

Hash Function: *Example*

For example : John Smith ==> sum of ascii values % 10
= (74+111+104+110+83+109+105+116+104) % 10
= 916 % 10
= 6 (index)



EMERTXE

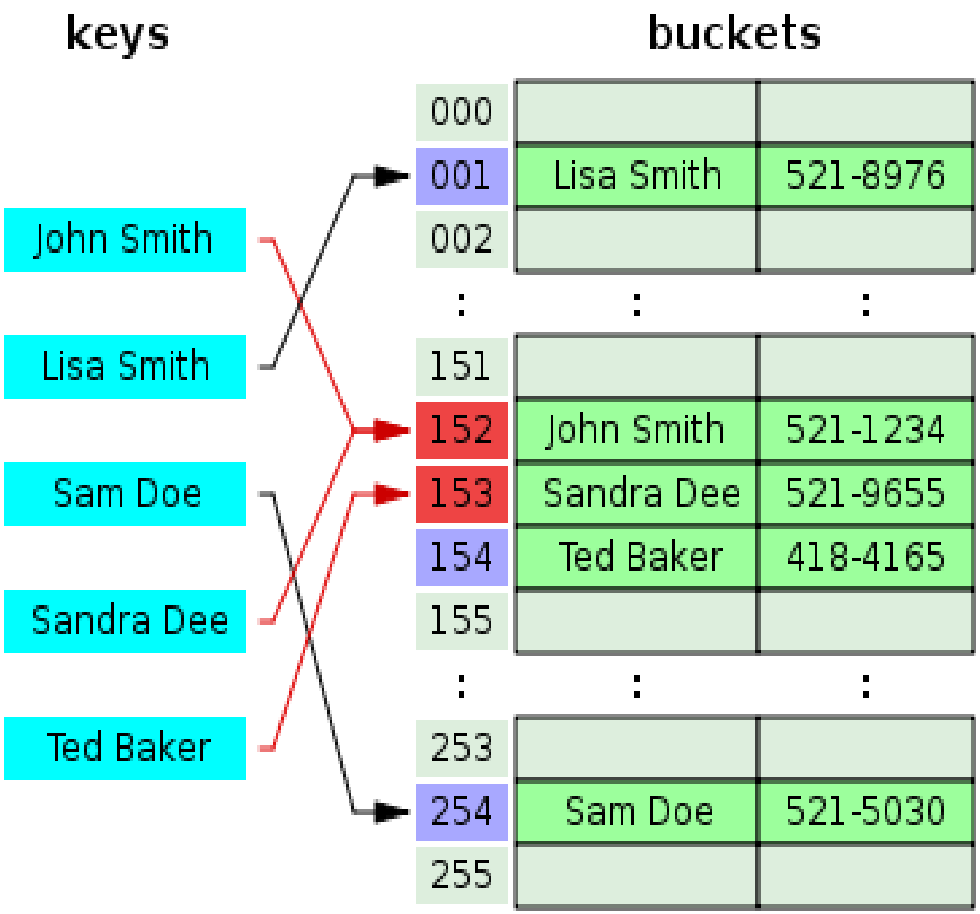
Collision Handling

- Ideally, the hash function should map each possible key to a unique slot index, but this ideal is rarely achievable in practice (unless the hash keys are fixed; i.e. new entries are never added to the table after it is created).

- Instead, most hash table designs assume that hash collisionsdifferent keys that map to the same hash valuewill occur and must be accommodated in some way.

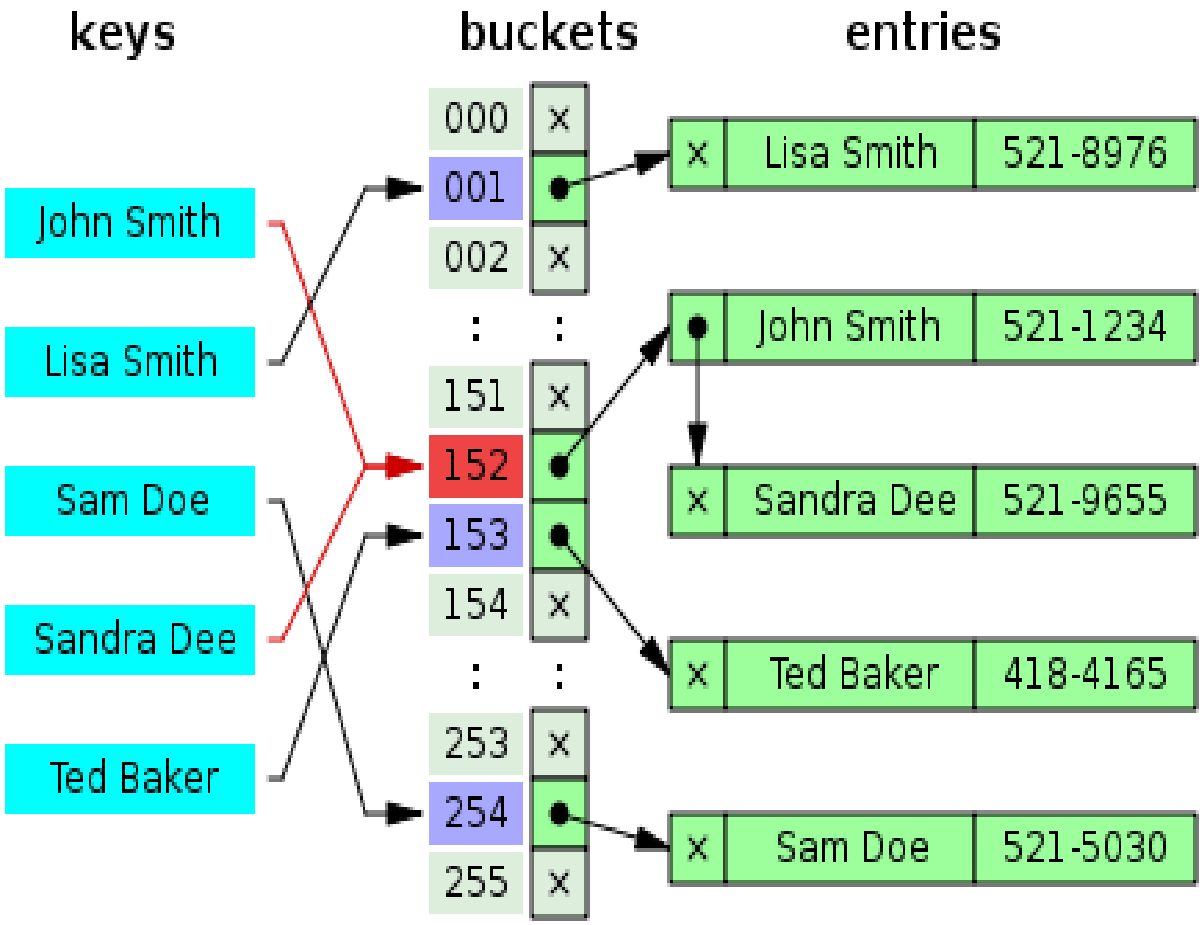
Collision Handling: Techniques

Open Addressing



Collision Handling: Techniques

Sepearte Chaining



ΣMERTXE

THANK YOU