



Topic: 4.1.2 Algorithms

An **algorithm** (pronounced AL-go-rith-um) is a procedure or formula for solving a problem. The word derives from the name of the mathematician, Mohammed ibn-Musa al-Khwarizmi, who was part of the royal court in Baghdad and who lived from about 780 to 850.

It is a step by step set of operations to be performed. It is almost as similar to a cooking recipe in real life. Algorithms exist that perform calculation, data processing, automated reasoning, sorting of data, and many other things.

In this chapter, we will be focusing on some of the different sorting and searching algorithms that exist today.

Linear Sort

This is the simplest kind of searching. It is also called the **serial sort** or the **sequential sort**. Searching starts with the first item and then moves to each item in turn until either a match is found or the search reaches the end of the data set with no match found.

A criteria will allow a possible match to be found within the records / items stored. If no match is found, then the process will return the appropriate message.

The Algorithm

1. Set up the search criteria
2. Examine first item in the data set
3. If there is a match, end the procedure and return the result with 'match found'
4. If no match is found repeat with the next item
5. If the last item is reached and no match is found return 'match not found'.

Advantages

Linear search is fairly easy to code. For example the pseudo-code below shows the algorithm in action.

```
procedure serial_search()
    For i = 0 to 19
        check_for_match(i, list)
        if match_found return 'match found'
    Next i
    return 'no match found'
end
```

Good performance over small to medium lists. Computers are now very powerful and so checking potentially every element in the list for a match may not be an issue with lists of moderate length.

The list does not need to be in any order. Other algorithms only work because they assume that the list is ordered in a certain way. Serial searching makes no assumption at all about the list so it will work just as well with a randomly arranged list as an ordered list.

Not affected by insertions and deletions. Some algorithms assume the list is ordered in a certain way. So if an item is inserted or deleted, the computer will need to re-order the list before that





Topic: 4.1.2 Algorithms

algorithm can be applied. The overhead of doing this may actually mean that serial searching performs better than other methods.

Disadvantages

 **May be too slow for oversized lists.** Computers take a finite amount of time to search each item, so naturally, the longer the list, the longer it will take to search using the serial method. The worst case being no match found and every item had to be checked.

This speed disadvantage is why other search methods have been developed.

Binary Search

This is a fast method of searching for an item in a **sorted / ordered list**.

Sometimes you may be doing a binary search without realizing it. Suppose you want to find Samuel Jones in the local telephone book. Would you start from page 1 and then go on from there, page by page? Unlikely.

You don't do this because you know an important fact about telephone books - the entries are in alphabetic order. So what you do is make a guess - J is about halfway down the alphabet and so you open the telephone book around half way. The page you see has names starting with N. So you know J will be in the first half of the book. Next you open a page about halfway down the first half - the page has 'H'. So now Jones must be in the upper half of this section.

You are carrying out a 'Binary search' algorithm. Notice that after only two guesses you are getting much closer to the answer. If you were carrying out a serial search, you would still be at page 2.

The Algorithm

1. Set the highest location in the list to be searched as N
2. Set the lowest location in the list to be searched as L
3. Examine the item at location $(N - L) / 2$ (i.e. halfway)
4. Is it a match? If Yes End search.
5. No
6. Is item less than criteria?
7. If Yes, Set lower limit L to item + 1 (Force the next search to use the upper half)
8. If No, Set upper limit N to item - 1 (Force the next search to use the lower half)
9. Is lower limit = upper limit, if yes end search (no match found)
10. Repeat from step 3 with the new upper and lower bounds.





Topic: 4.1.2 Algorithms

Is Binary searching better than serial searching?

It depends.

1. If the list is large and changing often, with items constantly being added or deleted, then the time it takes to constantly re-order the list to allow for a binary search might be longer than a simple serial search in the first place.
2. If the list is large and static e.g. telephone number database, then a binary search is very fast compared to linear search. (in math terms it takes $2\log_2(n)$ for a binary search over n items)
3. If the list is small then it might be simpler to just use a linear search
4. If the list is random, then linear is the only way
5. If the list is skewed so that the most often searched items are placed at the beginning, then on average, a linear search might be better.

If you have an array with N data items and you want to apply Linear and Binary search on this data set, The worst case will be if the item you want to look for is at the end of the array.

In this case a linear search would take N iterations to retrieve this item

Whereas, a binary search would take $2\log_2(n)$ iterations to retrieve this item.

To watch an animation for searching, click the link below:

<https://blog.penjee.com/wp-content/uploads/2015/04/binary-and-linear-search-animations.gif>





Topic: 4.1.2 Algorithms

Insertion Sort

In this method we compare each number in turn with the numbers before it in the list. We then insert the number into its correct position.

Consider the list

20 47 12 53 32 84 85 96 45 18

We start with the second number, 47, and compare it with the numbers preceding it. There is only one and it is less than 47, so no change in the order is made. We now compare the third number, 12, with its predecessors. 12 is less than 20 so 12 is inserted before 20 in the list to give the list

12 20 47 53 32 84 85 96 45 18

This is continued until the last number is inserted in its correct position. In Fig. 3.4.j.1 the blue numbers are the ones before the one we are trying to insert in the correct position. The red number is the one we are trying to insert.

20	47	12	53	32	84	85	96	45	18	Original list, start with second number.
20	47	12	53	32	84	85	96	45	18	No change needed.
			53	32	84	85	96	45	18	Now compare 12 with its predecessors.
12	20	47	53	32	84	85	96	45	18	Insert 12 before 20.
12	20	47	53	32	84	85	96	45	18	Move to next value.
12	20	47	53	32	84	85	96	45	18	53 is in the correct place.
12	20	47	53	32	84	85	96	45	18	Move to the next value.
12	20	32	47	53	84	85	96	45	18	Insert it between 20 and 47
12	20	32	47	53	84	85	96	45	18	Move to the next value.
12	20	32	47	53	84	85	96	45	18	84 is in the correct place.
12	20	32	47	53	84	85	96	45	18	Move to the next value.
12	20	32	47	53	84	85	96	45	18	85 is in the correct place.
12	20	32	47	53	84	85	96	45	18	Move to the next value.
12	20	32	47	53	84	85	96	45	18	96 is in the correct place.
12	20	32	47	53	84	85	96	45	18	Move to the next value.
12	20	32	45	47	53	84	85	96	18	Insert 45 between 32 and 47.
12	20	32	45	47	53	84	85	96	18	Move to the next value.
12	18	20	32	45	47	53	84	85	96	Insert 18 between 12 and 20.





Topic: 4.1.2 Algorithms

Algorithm

```
1 for j = 2 to length[A]
2     do key = A[j]
3         Insert A[j] into the sorted sequence A[1 . . j - 1].
4     i = j - 1
5     while i > 0 and A[i] > key
6         do A[i + 1] = A[i]
7             i = i - 1
8     A[i + 1] = key
```

Advantages of insert sort:

- Simple to code
- Very good performance with small lists (how small is small depends on the power of the computer it is running on)
- Very good when the list is almost sorted. In this case very few compares need to be done. The worst case is when the list is in reverse order.
- Sort-stable which means it keeps the relative positions of the elements intact
- Very memory efficient - it only needs one extra storage location to make room for the moving items.
- Good with sequential data that is being read in one at a time e.g. tape, hard disk or data being received sequentially.

Disadvantage of insertion sort compared to alternatives

- Poor performance with large lists.





Topic: 4.1.2 Algorithms

Bubble Sort

The Bubble sort is a simple sorting algorithm that works by repeatedly stepping through the list to be sorted, comparing each pair and swapping them if they are in the wrong order. The pass through the list is repeated until no swaps are needed, which indicated that the list is sorted. The algorithm gets its name from the way larger elements "bubble" to the top of the list. It is a very slow way of sorting data and rarely used in the industry. There are much faster sorting algorithms out there such as the Quick sort or the Heap sort.

Animation link :

http://en.wikibooks.org/wiki/A-level_Computing/AQA/Problem_Solving_Programming_Data_Representation_and_Practical_Exercise/Problem_Solving/Searching_and_sorting#/media/File:Bubble-sort.gif

Step-by-step example

Let us take the array of numbers "5 1 4 2 8", and sort the array from lowest number to greatest number using bubble sort algorithm. In each step, elements written in **bold** are being compared.

First Pass:

(**5** 1 4 2 8) → (**1** 5 4 2 8), Here, algorithm compares the first two elements, and swaps them since $5 > 1$
(**1** 5 4 2 8) → (1 **4** 5 2 8), It then compares the second and third items and swaps them 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.

The algorithm has reached the end of the list of numbers and the largest number, 8, has bubbled to the top. It now starts again.

Second Pass:

(**1** 4 2 5 8) → (**1** 4 2 5 8), no swap needed
(**1** 4 2 5 8) → (1 **2** 4 5 8), Swap since $4 > 2$
(1 2 **4** 5 8) → (1 2 4 5 8), no swap needed
(1 2 4 **5** 8) → (1 2 4 5 8), no swap needed

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.

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)

Finally, the array is sorted, and the algorithm can terminate.





Topic: 4.1.2 Algorithms

Algorithm

The algorithm can be expressed as:

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

It is an important point to note that the number of iterations that each algorithm takes to sort the data in either ascending or descending order may vary according to how the initial data is organized.

If the data is partially sorted, then the algorithms may take less iterations.

If the data is in descending order and you want to sort the data in ascending order, then the algorithm will take maximum iterations to sort the data.





Topic: 4.1.2 Algorithms

Linked List

Insertion

Consider Fig. 1a which shows a linked list and a free list. The linked list is created by removing cells from the front of the free list and inserting them in the correct position in the linked list.

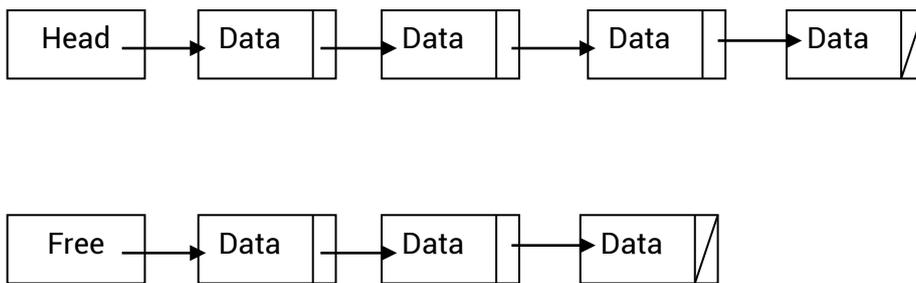


Fig. 1a

Now suppose we wish to insert an element between the second and third cells in the linked list. The pointers have to be changed to those in Fig. 1b

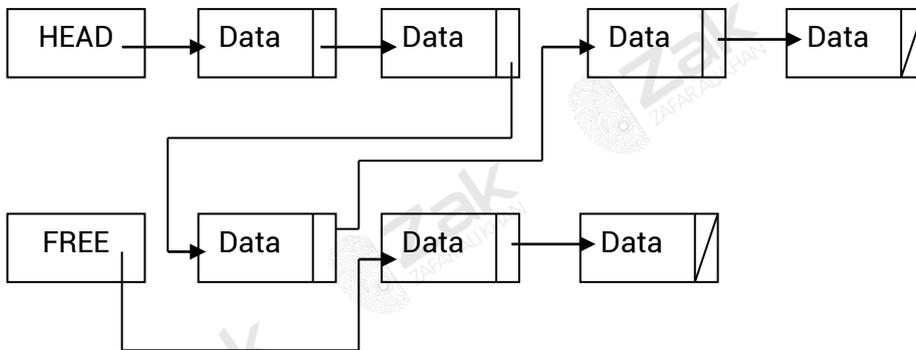


Fig. 1b

The algorithm must check for an empty free list as there is then no way of adding new data. It must also check to see if the new data is to be inserted at the front of the list. If neither of these is needed, the algorithm must search the list to find the position for the new data.





Topic: 4.1.2 Algorithms

The algorithm is given below.

1. Check that the free list is not empty.
2. If it is empty report an error and stop.
3. Set NEW to equal FREE.
4. Remove the node from the stack by setting FREE to pointer in cell pointed to by FREE.
5. Copy data into cell pointed to by NEW.
6. Check for an empty list by seeing if HEAD is NULL
7. If HEAD is NULL then
 - a. Pointer in cell pointed to by NEW is set to NULL
 - b. Set HEAD to NEW and stop.
8. If data is less than data in first cell THEN
 - a. Set pointer in cell pointed to by NEW to HEAD.
 - b. Set HEAD to NEW and stop
9. Search list sequentially until the cell found is the one immediately before the new cell that is to be inserted. Call this cell PREVIOUS.
10. Copy the pointer in PREVIOUS into TEMP.
11. Make the pointer in PREVIOUS equal to NEW
12. Make the pointer in the cell pointed to by NEW equal to TEMP and stop.

Deletion

Suppose we wish to delete the third cell in the linked list shown in Fig. 3.4.h.1. The result is shown in Fig. 1c.

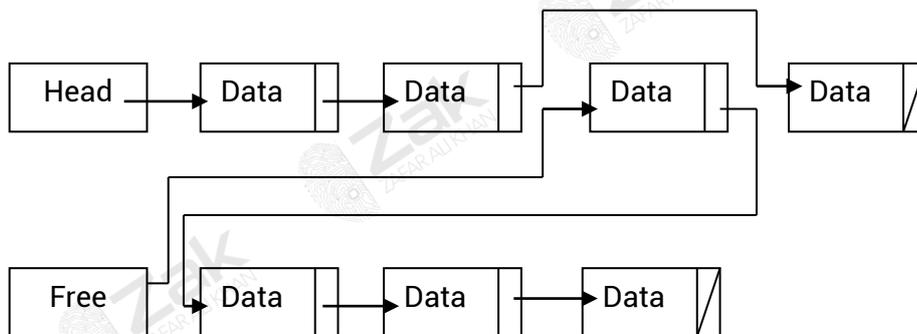


Fig. 1c

In this case, the algorithm must make sure that there is something in the list to delete.

1. Check that the list is not empty.
2. If the list is empty report an error and stop.
3. Search list to find the cell immediately before the cell to be deleted and call it PREVIOUS.
4. If the cell is not in the list, report an error and stop.
5. Set TEMP to pointer in PREVIOUS.
6. Set pointer in PREVIOUS equal to pointer in cell to be deleted.
7. Set pointer in cell to be deleted equal to FREE.
8. Set FREE equal to TEMP and stop.





Topic: 4.1.2 Algorithms

Amendment

Amendments can be done by searching the list to find the cell to be amended.

The algorithm is

1. Check that the list is not empty.
2. If the list is empty report an error and stop.
3. Search the list to find the cell to be amended.
4. Amend the data but do not change the key.
5. Stop.

Searching

Assuming that the data in a linked list is in ascending order of some key value, the following algorithm explains how to find where to insert a cell containing new data and keep the list in ascending order. It assumes that the list is not empty and the data is not to be inserted at the head of the list.

1. POINTER is equal to HEAD
2. NEXT is equal to POINTER.NEXT
3. DATA is equal to NEXT.DATA
4. While (newData is less than DATA)
 - a. POINTER is equal to NEXT
 - b. NEXT is equal to POINTER.NEXT
 - c. DATA is equal to NEXT.DATA
5. Endwhile

Note: A number of methods have been shown here to describe algorithms associated with linked lists. Any method is acceptable provided it explains the method. An algorithm does not have to be in pseudo code, indeed, the sensible way of explaining these types of algorithm is often by diagram.

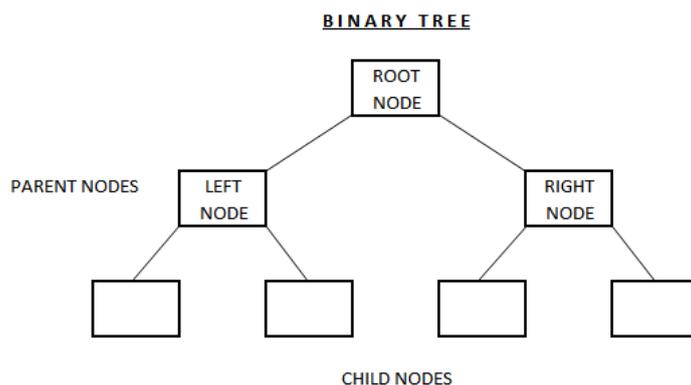




Topic: 4.1.2 Algorithms

Binary Tree

The TREE is a general data structure that describes the relationship between data items or 'nodes'. The parent node of a binary tree has only two child nodes.



Insertion - binary tree

One of the most powerful uses of the TREE data structure is to sort and manipulate data items. Most databases use the Tree concept as the basis of storing, searching and sorting its records. The Binary tree mostly holds data items in a sorted order, but with the addition of a simple rule

Rule: The LEFT node always contain values that come before the root node and the RIGHT node always contain values that come after the root node.

For numbers, this means the left sub-tree contains numbers less than the root and the right sub-tree contains numbers greater than the root. For words, as might be in a sorted dictionary, the order is alphabetic.

The algorithm for this tree creation is

1. If tree is empty enter data item at root and stop.
2. Current node = root.
3. Repeat steps 4 and 5 until current node is null.
4. If new data item is less than value at current node go left else go right.
5. Current node = node reached (null if no node).
6. Create new node and enter data.





Topic: 4.1.2 Algorithms

Example - forming a binary tree.

A sequence of numbers are to be formed into a binary search tree. These numbers are available in this order: 20, 17, 29, 22, 45, 9, 19.

Task: form a sorted binary tree diagram

This is done step by step.

Sequence 20, 17, 29, 22, 45, 9, 19

1. The first item is 20 and this is the root node, so begin the diagram

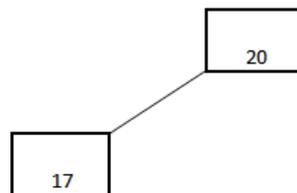
BINARY TREE



2. Sequence 20, 17, 29, 22, 45, 9, 19

This is a binary search tree, so there are two child nodes available, the LEFT and the RIGHT. The next number is 17, the rule is applied (left is less than parent node) and so it has to be the LEFT node, like this

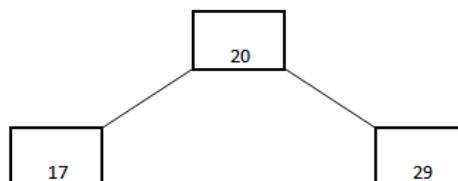
BINARY TREE

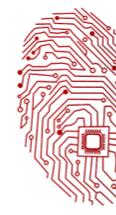


3. Sequence 20, 17, 29, 22, 45, 9, 19

The next number is 29, this is higher than the root node so it goes to the RIGHT sub-tree which happens to be empty at this stage, so the tree now looks like

BINARY TREE

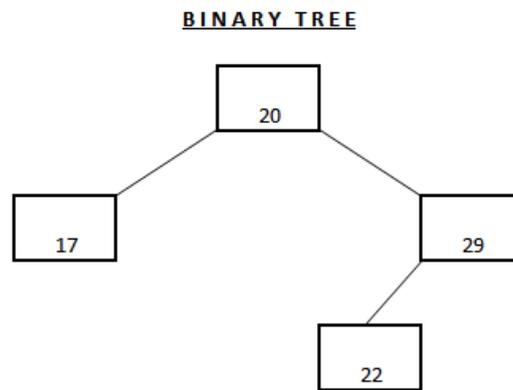




Topic: 4.1.2 Algorithms

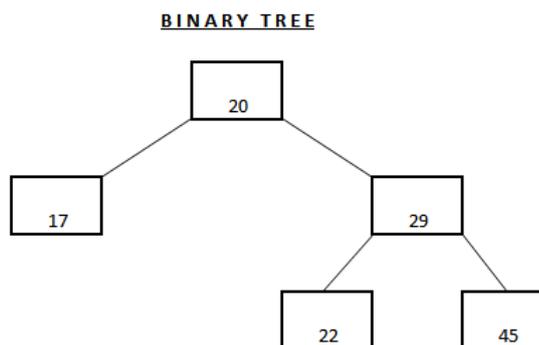
4. Sequence 20, 17, 29, 22, 45, 9, 19

The next number is 22. This is more than the root and so need to be on the RIGHT sub-tree. The first node is already occupied. So the rule is applied again to that node, 22 comes before 29 and so it needs to be on the LEFT sub-tree of that node, like this



5. Sequence 20, 17, 29, 22, 45, 9, 19

The next number is 45, this is more than the root and more than the first right node, so it is placed on the right side of the tree like this

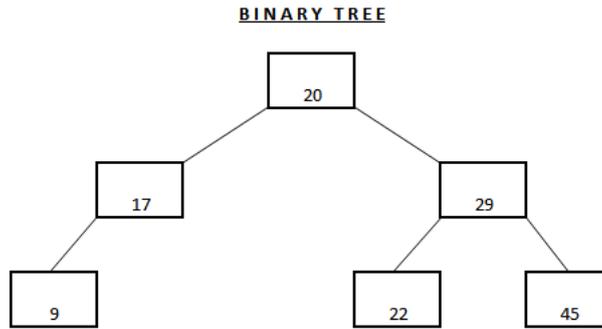




Topic: 4.1.2 Algorithms

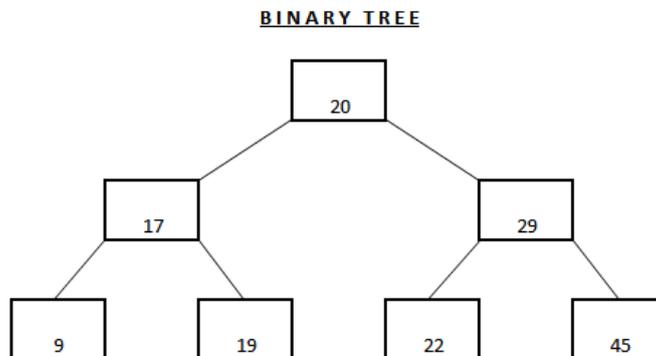
6. Sequence 20, 17, 29, 22, 45, 9, 19

The next number is 9 which is less than the root, the first left node is occupied and 9 is less than that node too, so it is placed on the left sub-tree, like this



7. Sequence 20, 17, 29, 22, 45, 9, 19

The next number is 19, which is less than the root, so it will need to be in the left sub-tree. It is greater than the occupied 17 node and so it is placed in the right sub-tree, like this



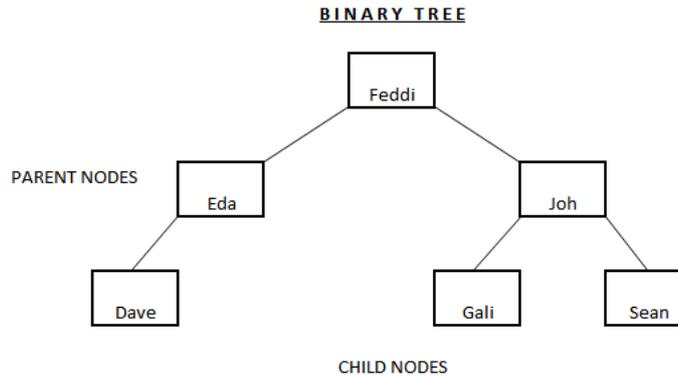
The tree is now complete.





Topic: 4.1.2 Algorithms

Deletion



Data in a tree serves two purposes (one is to be the data itself) the other is to act as a reference for the creation of the subtree below it. If Joh is deleted there is no way of knowing which direction to take at that node to find the details of the data beneath it.

Solution 1 is to store Joh's subtree in temporary storage and then rewrite it to the tree after Joh is deleted. (The effect is that one member of the subtree will take over from Joh as the root of that subtree).

Solution 2 is to mark data Joh as deleted so that the data no longer exists but it can maintain its action as an index for that part of the tree so that the subtree can be correctly negotiated.

Retrieval

The algorithm for this is

1. If tree is empty report error and stop.
2. Current node = root.
3. Repeat steps 4, 5 and 6 until program is stopped.
4. If required data item is equal to the value at current node then report FOUND and stop.
5. If required data item is less than value at current node then go left else go right.
6. If no node found then report error and stop.





Topic: 4.1.2 Algorithms

Stacks

Insertion

Fig. 3.4.h.4 shows a stack and its head pointer. Remember, a stack is a last-in-first-out (LIFO) data structure. If we are to insert an item into a stack we must first check that the stack is not full. Having done this we shall increment the pointer and then insert the new data item into the cell pointed to by the stack pointer. This method assumes that the cells are numbered from 1 upwards and that, when the stack is empty, the pointer is zero.

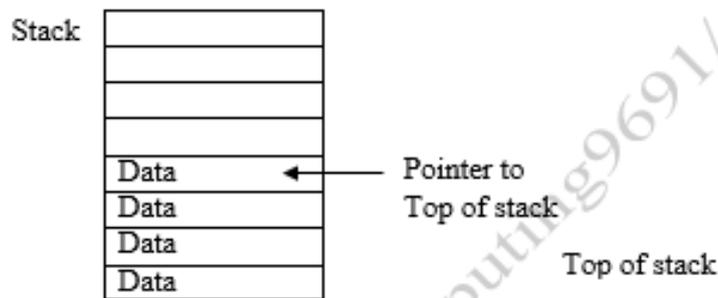


Fig. 3.4.h.4

The algorithm for insertion is

1. Check to see if stack is full.
2. If the stack is full report an error and stop.
3. Increment the stack pointer.
4. Insert new data item into cell pointed to by the stack pointer and stop.

Deletion (Equivalent to retrieval.)

When an item is deleted from a stack, the item's value is copied and the stack pointer is moved down one cell. The data itself is not deleted. This time, we must check that the stack is not empty before trying to delete an item.

The algorithm for deletion is

1. Check to see if the stack is empty.
2. If the stack is empty report an error and stop.
3. Copy data item in cell pointed to by the stack pointer.
4. Decrement the stack pointer and stop.

These are the only two operations you can perform on a stack.





Topic: 4.1.2 Algorithms

Queues

Fig. 3.4.h.5 shows a queue and its head and tail pointers. Remember, a queue is a first-in-first-out (FIFO) data structure. If we are to insert an item into a queue we must first check that the stack is not full. Having done this we shall increment the pointer and then insert the new data item into the cell pointed to by the head pointer. This method assumes that the cells are numbered from 1 upwards and that, when the queue is empty, the two pointers point to the same cell.

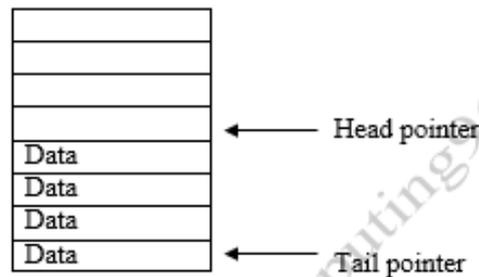


Fig. 3.4.h.5

The algorithm for insertion is

1. Check to see if queue is full.
2. If the queue is full report an error and stop.
3. Insert new data item into cell pointed to by the head pointer.
4. Increment the head pointer and stop.

Deletion (Equivalent to retrieval.)

Before trying to delete an item, we must check to see that the queue is not empty. Using the representation above, this will occur when the head and tail pointers point to the same cell.

The algorithm for deletion is

1. Check to see if the queue is empty.
2. If the queue is empty report error and stop.
3. Copy data item in cell pointed to by the tail pointer.
4. Increment tail pointer and stop.

These are the only two operations that can be performed on a queue.

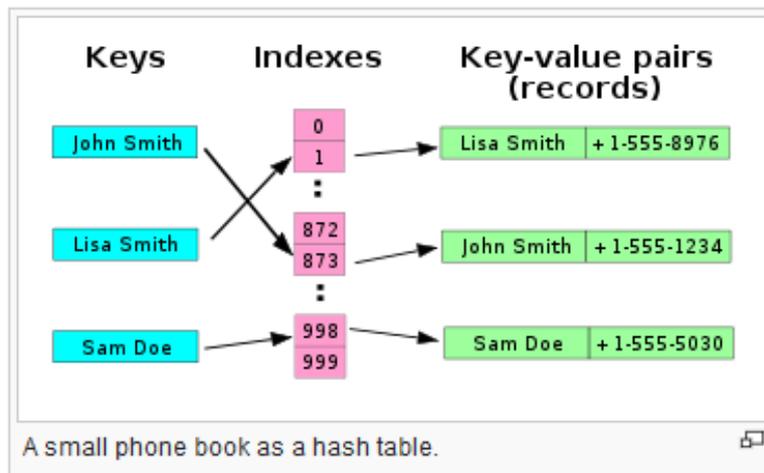




Topic: 4.1.2 Algorithms

Hash Table

A **hash table** or a **hash map** is a data structure that associates *keys* with *values*. The primary operation it supports efficiently is a *lookup*: given a key (e.g. a person's name), find the corresponding value (e.g. that person's telephone number). It works by transforming the key using a hash function into a *hash*, a number that the hash table uses to locate the desired value.



Insertion

In a hash table, Each slot either contains a key or NIL (if the slot is empty) and if the slot is empty, then you can insert your data item in that slot. If the hash table is searched until the end, and no empty slots are found, then the algorithm will return "hash table full"

```
hash_insert (T, k)
i := 0
repeat j := h(k, i)
if T[j] = NIL
then T[j] := k
else i := i + 1
until i = m
error "hash table overflow"
```

Searching

```
hash_search(T, k)
i := 0
repeat j := h(k, i)
if T[j] = k
then return j
i := i + 1
until T[j] = NIL or i = m
return NIL
```





Topic: 4.1.2 Algorithms

For different searching and sorting algorithms, the logic behind their functioning differs. Some algorithms can return the answer in less number of iterations than compared to other algorithms.

If we compare the Binary search to the Linear search, it should be quite obvious by now that the Binary search takes fewer iterations to return the value being searched for. However, as mentioned before, the Linear search is very easy to implement which means that it does not take up a lot of computer memory for its functioning unlike the Binary search. This is the compromise that you have to make for stronger algorithms. Those which take fewer iterations (take less time to return answer) are more likely to consume more computer memory and vice versa...

