Array

100	104	108	112
10	20	30	40

- 논리적 저장 순서, 물리적 저장 순서 일치
  - 데이터 메모리 저장시, 저장 시작 위치부터 빈자리 없이 순서대로 저장.
  - index로 element에 접근 가능, index 알고 있으면 O(1) 접근 - random access
  - insertion, deletion : 접근하여 작업 완료 O(1) + α (shifting)

· collection of similar types of data stored at contiguous memory locations.

· Simplest data structure where the data element can be accessed randomly using index number

multidimensional array

· DS that span across more than one dimension

- more than one index variable for every point of storage - 2D array : emulates the tabular from structure which provides ease of holding the bulk of data that are accessed using row and column pointers.

Static	
lookup	O(1)
push	O(1)
insert	O(n)
delete	O(n)

dynamic	
lookup	O(1)
append	O(1) or O(n)*
insert	O(n)
delete	O(n)

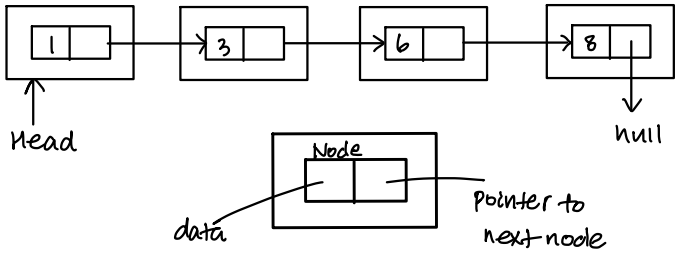
* expanding memory

Linked list

chainlike

· sequence of nodes where every node is connected to the next node by a reference pointer. The elements are NOT stored in adjacent memory location. → chain

· two parts : data field, reference to the next node.



prepend	O(1)
append	O(1)
lookup	O(n)
insert	O(n)
delete	O(n)

확인!

1. Are linked list linear or nonlinear type?
- linear : used for access strategies
nonlinear : used for data storage.

access strategies vs. data storage

2. How are linked lists more efficient than arrays?
① insertion & deletion : room has to be created + shifted (array) / update the address present in the next pointer. (LL)
② dynamic DS : LL - dynamic, don't need to give initial size, can grow/shrink at runtime by alloc. + dealloc.
array - limited as # of items is statically stored in main memory.
③ no wasting memory : LL - grow + shrink based on the needs, wasted X
array - declaring size of 10 and storing 3, wasted

* array는 메모리를 alloc을 하고, LL는 pointer가 하나 더 있음.
* random access

array는 lookup에서 efficient → index를 알고있으면 random access 가능.

3. Scenario where you can use array, linked list.

① linkedlist:

i) when we don't know exact # of element
ii) if there are a lot of insertion + deletion.
iii) less # of random access operation.
iv) inserting items anywhere in the middle of list, like implementing priority queue

② Array:

i) index | random access elements more frequently.
ii) knowing # of elements in the array beforehand to allocate the right amount of memory
iii) speed iterating over the elements in the sequence
iv) memory - when filled array use less memory
↳ represents data + pointers

4. Doubly linked list, applications.

- two references - next node, previous node.
- traversal of the data elements in both direction
- Applications
  - i) a music playlist with next song and previous song
  - ii) browser cache with BACK - FORWARD visited page.
  - iii) undo/redo functionality

5. Process behind storing a variable in memory.

- stored based on the amount of memory that is needed.
  - i) required amount of memory is assigned first
  - ii) stored based on the DS being used
    - dynamic alloc. ensures high efficiency + storage units can be accessed based on requirements.

6. How to implement a queue using stack?

- two stacks - stack1, stack2, queue
  - stack: push, pop, peek.
  - queue: enqueue, dequeue

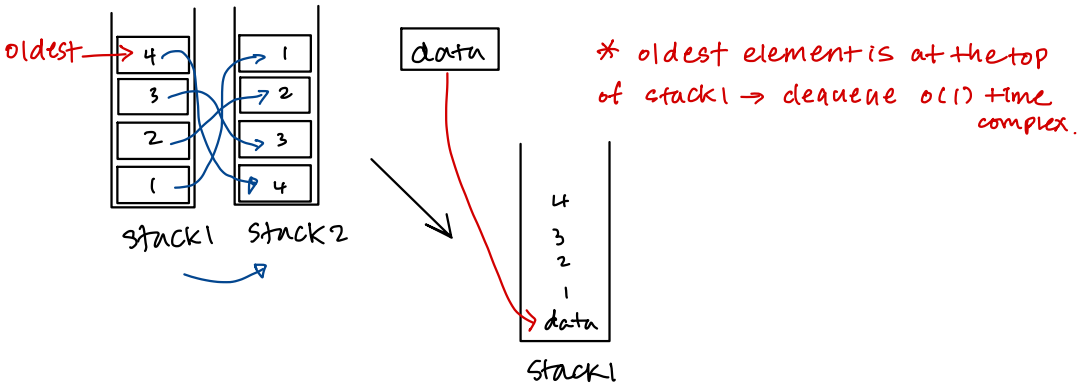
① making enqueue operation costly

i) enqueue(q, data): O(n)

```
while stack1 not empty:
    push everything stack1 to stack2
    push data to stack1
    push everything back to stack1
```

ii) dequeue(q): O(1)

```
if stack1 is empty then error
else pop an item from stack1 and return.
```



② making dequeue operation costly

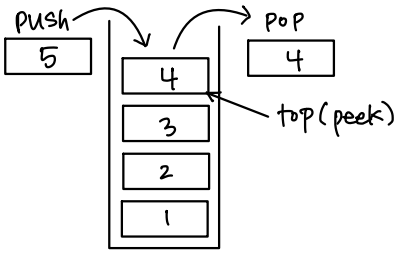
enqueue O(1),
push data to stack1

dequeue O(n)
if both stacks empty → error
if stack2 empty
while stack1 ! empty
push all stack1 → stack2
Pop the element from stack2, return

StackOverflow? How, why, when?
Array Stack, LinkedList stack

Stacks

· linear DS that follows LIFO approach for accessing.



applications
- checking for balanced parentheses
- evaluation of postfix expression.
- problem of infix to postfix conversion
- reverse a string.

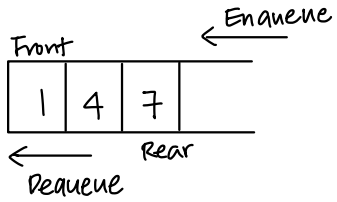
lookup O(n)
pop O(1)
push O(1)
peek O(1)

stack 이용 이중괄호 구현 stack 두개로 Queue 구현
Queue 사용 Heap 구현 stack으로 깊은 우선탐색 구현

Array Queue, LinkedList Queue
FAQ: Queue using stack.

Queues

· linear DS that follows FIFO approach for accessing element.



applications
- CPU Task scheduling.
- BFS algorithm to find shortest distance b/w two nodes in graph.
- Website nearest processing.
- used as buffers in applications MP3, CD.
- managing an input stream.

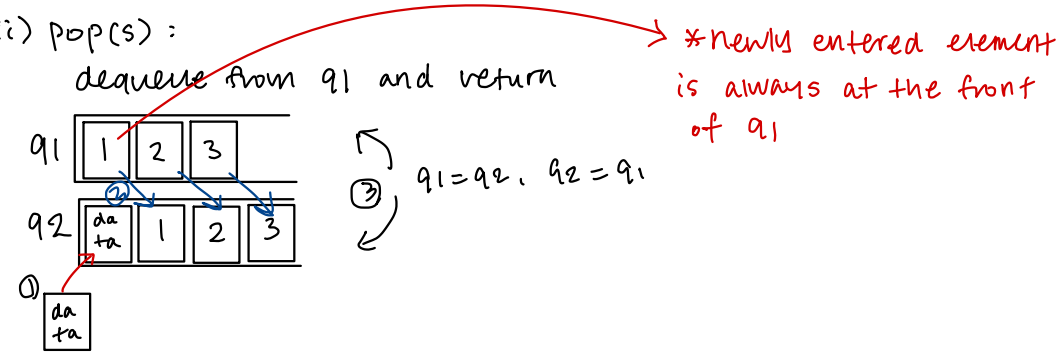
lookup O(n)
enqueue O(1)
dequeue O(1)
peek O(1)

7. How do you implement stack using queues?

① making push operation costly.

i) push(s, data):
enqueue data to q2
dequeue elements one by one from q1, enqueue to q2
swap the names of q1 and q2

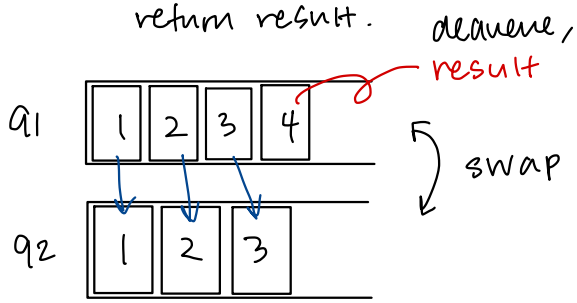
ii) pop(s):
dequeue from q1 and return



② making pop operation costly

i) push(s, data):
enque data to q1

ii) pop(s):
dequeue all except the last elem. from q1, enqueue q2
dequeue last item q1, dequeue item stored in result variable.
swap q1, q2
return result.



Hash Table

array (indexing)

배열을 사용하여 data 저장하므로 - fast searching by index O(1) Time (average)

but problem : index로 저장되는 key 값이 보장되지 않. so hash function. (unique)

Hash Function : unique index 설정 (hash code). 저장되는 값들의 key 값을 hash func을 통해 작은 값들로 바뀌줌. (can be same hash code - collision)

- collision : 서로 다른 두개의 키가 같은 index (hash code) → duplicate.
- 키 전체를 참조하여 hash 값을 만들어 냄
- collision을 처리하는 방법도 선택 (1:1 X) → array랑 다르게 X, 메모리↑, collision 대비
- collision ↑ → search time complexity O(1) → O(n)

Resolving Collision (느낌) — bucket 밀도↑ → worst case 발생 빈도↑

1. open address : collision이 일어나면 다른 hash bucket에 insert.
   - ① linear probing - 순차적으로 탐색하여 비어있는 bucket 찾기.
   - ② quadratic probing - 2차함수를 이용해 탐색할 위치 찾기.
   - ③ double hashing probing - 하나의 hash func collision → 2차 hash func (다만든 연산량 ↑)
2. separate chaining : 보조 hash func → Java의 HashMap. (hash값 충돌 가능성↓, 처리위한 연산적 처리시간 + 메모리)
   - ① Linked List : bucket을 linked list 만들어 collision → bucket에 list 추가. 삭제/삽입 간단, 연속 공간 저장시 overhead 부담. → 데이터 확장 늦춤.
   - ② Red Black Tree : hash bucket에 할당된 key-value 쌍의 개수. ↓ - LL, ↑ - RBT. 메모리 사용↑

open address vs separate chaining : both worst case O(M)

open address : 연속된 공간에 data 저장 → cache 효율↑ → 데이터 개수 적다면.

bucket 계속 사용 → separate chaining 테이블 확장 늦춤.

hash bucket resizing : HashMap은 key-value 쌍 데이터 개수가 일정 개수가 되면 bucket 개수를 두배로 늘림 → collision 성능 손실 문제 해결

when? 데이터 개수가 hash bucket 개수의 75%. load factor : 0.75.

8. What is hashmap in data structure?

: uses implementation of hash table DS, allows access of data in constant time O(1) if you have a key.

9. What is requirement for an obj to be used as key/value in Hashmap?

- key/value obj used in hashmap must imple. equals(), hashcode()
- hashcode is used when inserting the key obj into the map. equals ——// retrieving a value from the map.

10. How does HashMap handles collision in Java?

Chaining : if new values with same key are attempted to be pushed, then these values are stored in a linkedlist stored in bucket of the key as a chain along with the existing value.

* worst case : having same hashcode → turning linkedlist searching O(n)

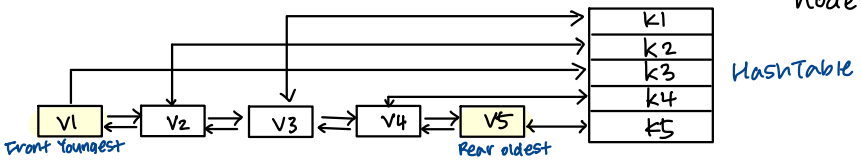
Space	O(n)
insert	O(1)
lookup	O(1)
delete	O(1)

could be O(n) with hash collisions and dynamic array resizing but unlikely.

11. Which DS are used for implementing LRU Caches? (Least Recently Used)

LRU cache allows quick identif. of an element that hasn't put to use for the longest time by organizing items in order of use

- queue : imp. using DLL. max size of the queue is determined by the cache size ex) total # of available frames.
  LRU pages will be near the front end, most recent → near end
- hashmap : stores page # as key along with address corr. queue node as value.



Tree : nonlinear DS – Hierarchical relationship representation.

- node : tree 구성의 요소.
- edge : tree 구성 위해 node를 연결하는 선.
- root node : 최상위 node
- terminal node (leaf node) : 하위에 다른 node가 연결되어있지 않는 node
- internal node : terminal node 제외 모든 node, root node 포함.

Binary Tree : root node 중심으로 두개의 subtree로 나뉘짐. 나뉘진 두 subtree도 모두 BT 이야함. 공집합도 BT, node 하나도 BT (by induction) — extension of linked list

- level : root의 node level은 0, 최고 level은 height
- perfect BT (포화), complete BT (완전), Full BT (정) — differences?

모든 level이 꽉 찬 BT / 위→아래, 왼→오 순서대로 채워진 BT / 모든 node가 0개 혹은 2개의 자식 node를 갖는 BT

index =

- parent(i) = i/2, left_child(i) = 2i, right_child(i) = 2i+1 (node의 개수 n, root i = 1)

Binary Search Tree — O(log n) / O(h) (h가 높아질수록 추가할 수 있는 node의 수가 두배씩 증가.) — skewed tree (편향) – O(n)

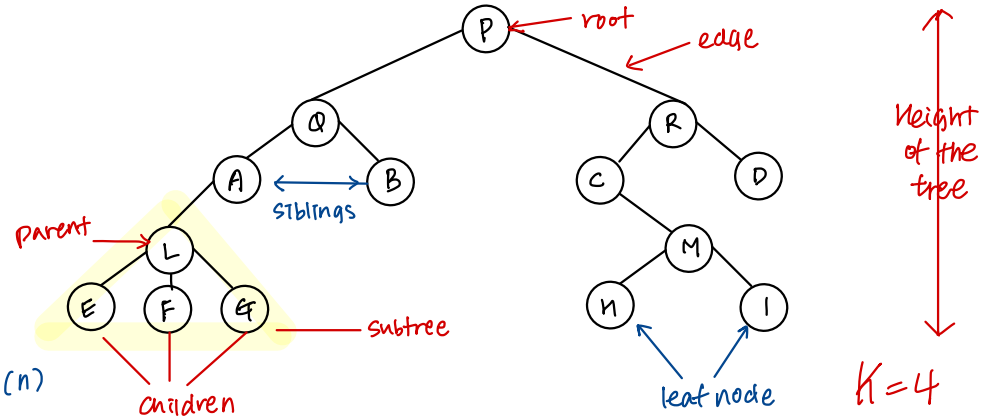
rule : ① BST node key는 unique
② Parent node key > left child node key
③ Parent node key < right child node key
④ left, right subtree → BST

???

Binary Heap : 배열에 기반한 Complete BT. 1번 index부터 시작. — node의 고유번호 값과 index 일치시킴.

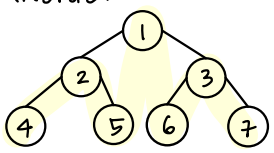
- max heap : 각 node의 값이 해당 children의 값보다 크거나 같음.
  root node value 가장 큼 → max value search O(1)
  heapify : heap의 구조를 계속 유지하기 위해서 제거된 node는 대체할 다른 node 필요 → 맨 마지막 node를 root node로 대체 시킴 O(log(n))

- Tree is recursive, non-linear DS consisting the set of one or more data nodes where one node is designated as root, remaining node children
- organizes data into hierarchical manner
- most commonly used tree DS is a binary tree and its variants
- applications :
  ① Filesystem – files inside folders inturn inside other folders
  ② comments on SNS – comments, replies
  ③ Family trees – parents, grandparents, family hierarchy.

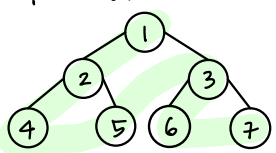


12. what is max # of nodes in a binary tree of height K?
$2^{K+1}-1$ where $K \geq 1$

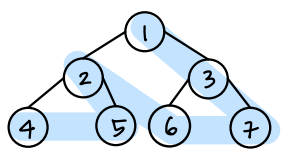
13. what are tree traversals? : process of visiting all nodes.



inorder
left – root – right
4 → 2 → 5 → 1 → 6 → 3 → 7
BST : gives nodes in ascending order



preorder
root – left – right
1 → 2 → 4 → 5 → 3 → 6 → 7
used to create copy
Prefix expression of an expression tree



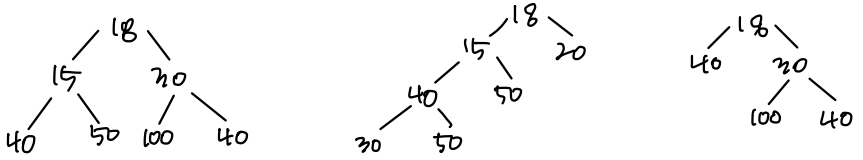
Postorder
left – right – root
4 → 5 → 2 → 6 → 7 → 3 → 1
used to delete tree
Postfix expression of an expression tree.

Why?

Balanced BST	
lookup	O(logn)
insert	O(logn)
delete	O(logn)

Binary Heap	
lookup	O(n)
insert	O(logn)
delete	O(logn)

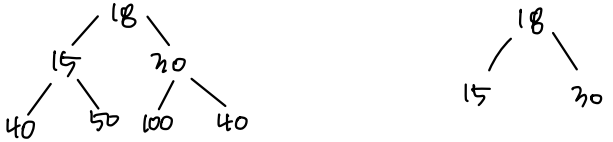
- full BT: every node has 0 or 2 children



- Complete BT: all levels are completely filled except possibly the last level and the last level has all keys as left as possible.

practical example : binary heap -

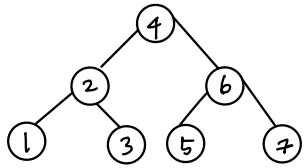
- Perfect BT: all the internal nodes have two children and all leaf nodes are at the same level



- Balanced BT: height of the tree is O(log n), where n: # nodes. AVL maintains o(log n) height $-|\text{height left} - \text{right}| \leq 1$

- Degenerate tree: every internal node has one child. (pathological) performance-wise as linked list.

14. BST: variant of BT in efficient way that values of nodes in the left subtree are less than root node, values on right root node are higher than root node



15. AVL Tree: height balancing - checks the height of left + right subtrees and assures that the difference is not more than 1.

Difference is called balance factor = height (left subtree) − height (right subtree)

16. Heap DS: tree based, nonlinear DS, complete BT. (all levels are completely filled except last level, last level has all elements toward as left as possible)

- max heap - root node must be greatest among all the data elements
- min heap - // smallest //

17. Advantage of heap over stack?

- heap is more flexible than the stack - memory space for heap can be alloc/ dealloc as needed.
- heap can at times be slower.

18. priority queue

- abstract data type like queue but has priority assigned to elements.
- Elements with high priority are processed before a lower priority.
- min 2 queues are required i) data ii) store priority.

linear vs. nonlinear

linear: ① data elements arranged in a linear order where every elements are attached to its prev, next adjacent.
② single level involved
③ implementation is easy in comparison
④ data elements can be traversed in a single run only
⑤ memory is not utilized in an efficient way
⑥ array, stack, queue, linked list

nonlinear: ① data element attached in hierarchically manner
② multilevel involved
③ impl. complex.
④ data elements can't be traversed in a single run only
⑤ memory is utilized in a efficient way
⑥ trees & graph.

heap
- complete BT, priority queue 위해 만들어진 DS.
- 여러개의 값들 중 max, min 바로 찾아내도록 만들어짐.
- 방법론 상태 ㄴ 큰 값이 상위 레벨, 작은 값 하위 (max heap)
  ㄴ 부모의 값이 자식 값보다 항상 작은 BT

Graph: nonlinear DS consist of a definite quantity of vertices and edges. The vertices or the nodes are involved in storing data and the edges show the vertice relationship. The difference between a graph to a tree is that in a graph there are no specific rules for the connection of nodes. ex) SNS, telephone netw. can be represented through the graph.

Stack: stores temporary variables created by a func. Var declared, stored and initialized during runtime.

heap: stores global variables. supports dynamic memory alloc. Not managed automatically, tightly managed by CPU.
Every time when we made an object it always creates in heap-space and the referencing info to these objects are always stored in stack-memory. Heap memory alloc. isn't safe as stack memory alloc. was because data stored in this space is accessible/visible to all threads. → memory leak.

① young generation: all new data(obj) are made to alloc. the space and whenever memory is completely filled, rest → garbage collec
② old/tenured generation: older data obj that are not in freq. use/not in use at all are placed.
③ permanent generation: contains JVM's metadata for the runtime classes and application methods

1. what is data structure?
: refers to the way data is organized and manipulated.
Seeks to find ways to make data access more efficient.

2. Differentiate between files and storage structure.
DS is memory area that is being accessed
- resides main memory of computer sys. : storage structure
when dealing with auxiliary structure : file structure.

3. when is binary search best applied?
: best applied to search a list when elements are already in order (sorted). searched in the middle O(log(n))

4. where do data structure are applied?
: algorithms that involve efficient data struc.
ex) numerical analysis, operating system, AI, compiler design, database management, graphics, statistical analysis.

5. what DS are applied when dealing with a recursive func?
- recursion - func calls itself based on terminating condition.
- using LIFO, a call to a recursive func saves the return address so that it knows how to return to the calling func after call terminates.

6. How does dynamic memory alloc. help in managing data?
: can combine separately allocated structured blocks to form composite structure that expand and contract as needed.

7. ordered list
: each node's position in the list is determined by the value of its key component. key values form an increasing seq as traversed.

8. merge sort
: divide-and-conquer - adjacent ones are merged and sorted to create bigger sorted list until one single list.

9. NULL vs VOID
- Null - value : indicates an empty value
- void - data type identifier : identifies pointers as having no initial size.

10. How does variable declaration affect memory alloc?
: allocated / reversed depend on the data type of var. declared.
ex) integer type - 32 bits (4 byte) reversed for var.

11. postfix expression
: each operator follows its operands
adv - no need to group sub-expressions in parentheses / operator precedence

12. data abstraction
: breaking down complex data problems in manageable chunks.
Applied by initially specifying data objects involved and the operation to be performed without being concerned how they are represented + stored in memory.

13. selection sort
smallest element first located and switched with the element at subscript zero

14. signed vs. unsigned affect in memory?

signed: first bit is used to indicate whether positive / negative, one bit short.

unsigned: all bits available for the number

unsigned 8 bit 0 - 255
signed 8 bit -128 - 127

15. what is min number of nodes a BT can have?

BT can have min 0 nodes → when nodes have NULL values
and BT can also have 1 or 2 nodes

16. Dynamic data structure.

DS that expand and contract as a program run - flexible size

17. Pointers applied DS

pointers used in linked list - stack, queue, linked list, BT

18. Do all declaration stat. result in a fixed reserv. in memory?

most declaration do. with the exemption of pointers.
Pointer decl. does not alloc. memory for the data comes during runtime.

19. min # of queues needed when implementing priority queue?

2. one queue intended for sorting priorities
other queue used for actual storage of data

20. fastest sorting algorithm?

No one. bic each algorithm is designed for a particular DS and data set. Depends on data.

ex) quick sort, bubble sort, balloon sort, radix sort merge sort, etc.

21. Basic algorithm BST.

① empty → target is not in tree → end search.
② not empty - check the target in the root
not root, check smaller or bigger than root
smaller → check left subtree
bigger → check right subtree

22. Deqeue?

double-ended queue. elements can be inserted / removed either end.

23. bubble sort?

can be applied to array. Comparing adjacent elements and exchange their values → lets smaller values "bubble" to top larger value sink to bottom.

24. parts of linked list?

head and tail. actual node between. linked sequentially.

25. graph?

contains a set of ordered pairs - edge (arc): connect nodes

26. selection sort?

picking smallest # from the list, placing front. first → end.
simplest sorting algorithm.

27. linear vs. nonlinear

linear: data elements are adjacent each other
ex) array, linked list, stacks, queue.

nonlinear: each data element can connect to more than two adjacent elements ex) graph, tree.

28. AVL tree
BST that always in a state of partially balanced.
Balance is measured as diff. b/w heights of subtree from the root. self-balancing tree known to be 1st to be designed.

29. Doubly linked list
traversal across the data elements can be done in both directions. Two links everynode → previous, next node.

30. Huffman's algorithms.
creating extended BT that have min. weighted path lengths from the given weights. use of table contains the frequency of occurrence for each data element.

31. Fibonnacci search?
Divide-and-conquer approach - reduce the time needed in order to reach the target.

32. recursive algorithm?
Dividing problem into smaller, manageable sub-problems.
The output of one recursion after processing one subprob. becomes the input to the next recursive process.

33. searching for a target key in linked list.
Sequential search - each node is traversed / compared w/ target key ... continues until either the target key is found / last node is reached.

Heap is used to store dynamic variables: region of process's memory : calloc(), malloc(), resize()

heap overflow
① allocate dynamic large number of variables
② continuously allocate memory and do not free after using it

Stack is LIFO. used to store local variables which is used inside the function. Parameters are passed through this function and their return addresses.
If program consumes more memory space, stack size is limited in computer memory
① function called recursively by itself infinite times then stack will be unable to store large number of local variables
② if we declare a large number of local variables or declare a large dimensional array

34. function vs. method
function - piece of code that is called by name. It can be passed data to operate on and can return data. All data that is passed to a function is explicitly passed.
method - piece of code that is called by name that is associated with an object.
① implicitly passed the object on which it was called.
② able to operate on data that is contained within the class

standalone feature or functionality
associated with an object.