Data structures (229)

/**

For this chapter I decided not to screenshot the Scala implementations. Instead visit my GitHub link: https://github.com/FelixChen78/Data_structures

*/

Main functionality of a set:

1. Query: return information from set
2. Modifying operations: changes the set

Typical operations of a set:

SEARCH(S, k)
A query that, given a set S and a key value k, returns a pointer x to an element in S such that $x.key = k$, or NIL if no such element belongs to S.

INSERT(S, x)
A modifying operation that augments the set S with the element pointed to by x. We usually assume that any attributes in element x needed by the set implementation have already been initialized.

DELETE(S, x)
A modifying operation that, given a pointer x to an element in the set S, removes x from S. (Note that this operation takes a pointer to an element x, not a key value.)

MINIMUM(S)
A query on a totally ordered set S that returns a pointer to the element of S with the smallest key.

MAXIMUM(S)
A query on a totally ordered set S that returns a pointer to the element of S with the largest key.

SUCCESSOR(S, x)
A query that, given an element x whose key is from a totally ordered set S, returns a pointer to the next larger element in S, or NIL if x is the maximum element.

PREDECESSOR(S, x)
A query that, given an element x whose key is from a totally ordered set S, returns a pointer to the next smaller element in S, or NIL if x is the minimum element.

Stack and queues:

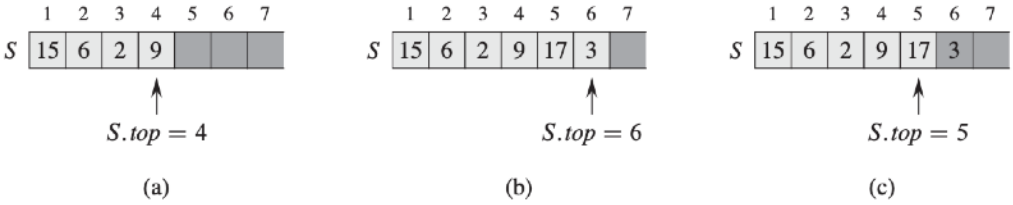
Overflow: When trying to insert into stack or queue when it is full

Underflow: When trying to pop/ dequeue from empty stack or queue

Stack: LIFO (like a stack of plates, most recent plate topped first to remove)

Two operations:

1. Push O(1)
2. Pop O(1)



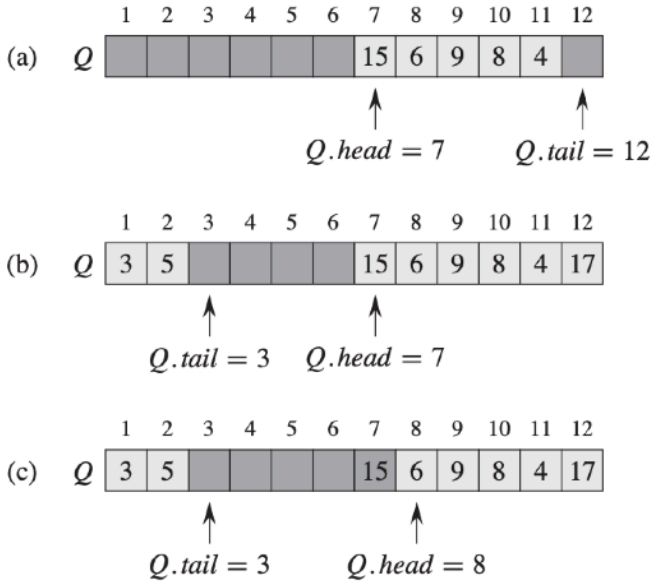
There are 4 elements in (a) and the top value is 9. When it goes to (b), 17 is pushed on then 3 is pushed on. There are 6 elements and the top value is 3. When it goes to (c), 3 is popped from the stack leaving 5 elements and a top value of 17.

Array based stack Scala implementation:

** Could also be implemented using linked list by storing top value in a pointer

Queue: FIFO (like a line, first in line first serve)

When head = tail + 1 then queue is full. Attempting to enqueue will cause queue overflow



The diagrams uses head and tail to illustrate the start and empty slot next to the of the queue. For (a) the head starts at the 7th slot of the array and the tail ends at the last slot of the array. For (b), 17, 3, 5 are enqueue in this order onto the queue and the new tail position is the 3rd slot. For (c), the queue dequeues so the new head position is the 8th slot while the 7th slot value is returned which is 15.

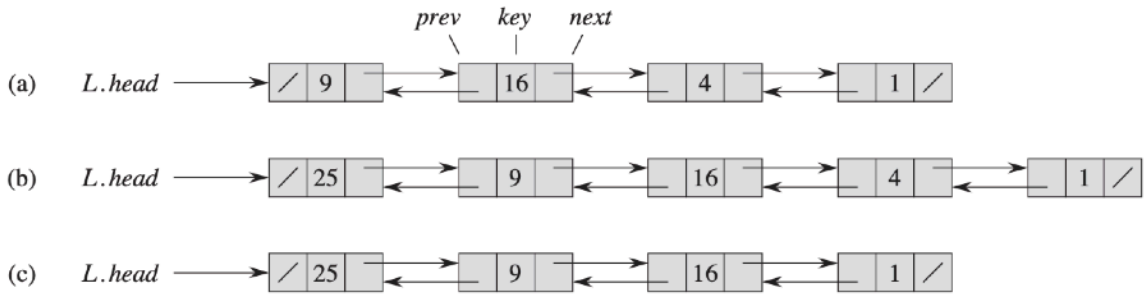
** A circular linked list can be used to implement a queue too since the head and tail are easily accessible.

Linked List: Unlike array which is oranges by linear order (indices), linked list order is determine by pointers for each object. A doubly linked list has two pointers: prev and next. Prev points to the predecessor and next points to the successor of the node. If prev is null/ nil, it means the current node is the first node/ head of the list.

Variations of linked List(sorted/ unsorted):

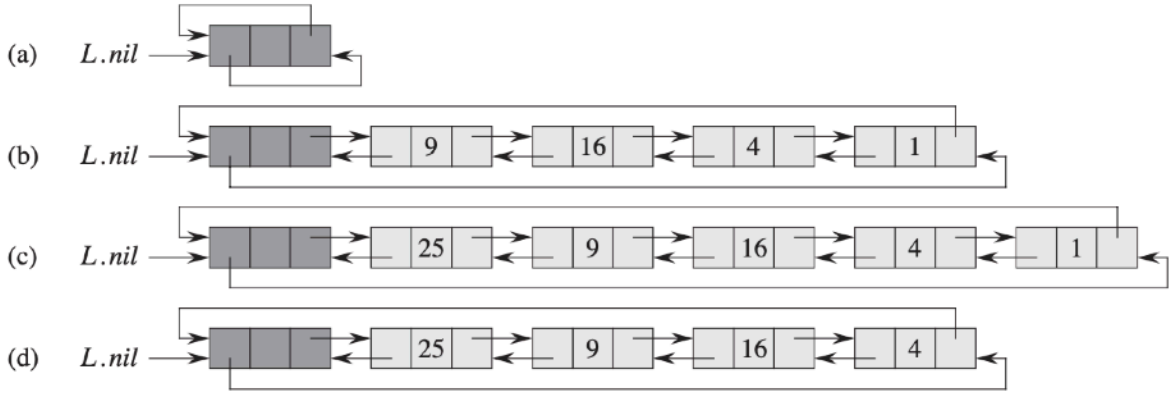
1. Singly: one pointer
2. Doubly: two pointers
3. Circular: one pointer; tail points to head

Doubly linked list



Sentinel: is a dummy object that allows to simplify boundary conditions. It is mainly used to create clearer code rather than increasing speed as it only reduces the coefficient of runtime.

Circular doubly linked list

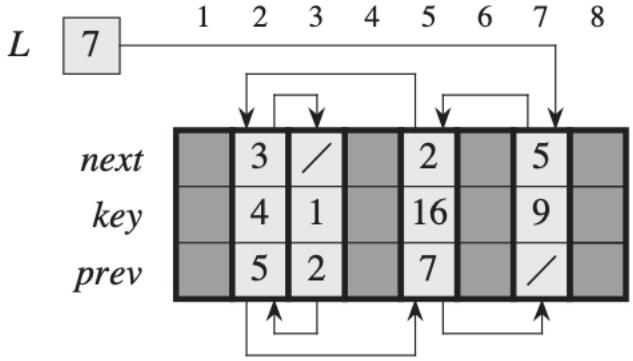


In a circular doubly linked list, the head and the tail both point to the sentinel

Pointer and objects

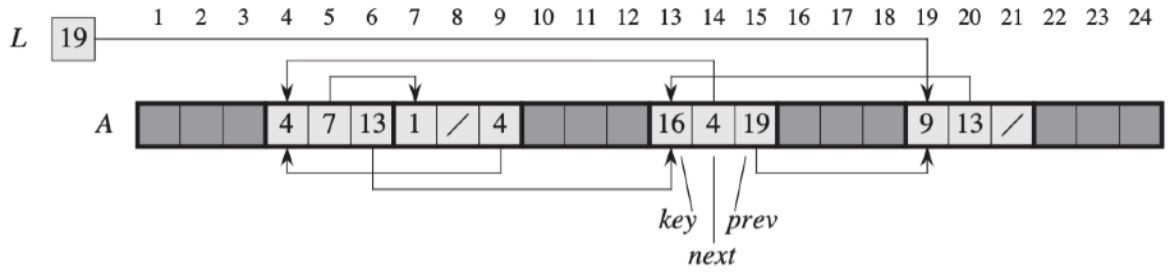
Use array and array indices

Multi-array



In the diagram, the pointer is simply a common index for key, next, prev

Single-array

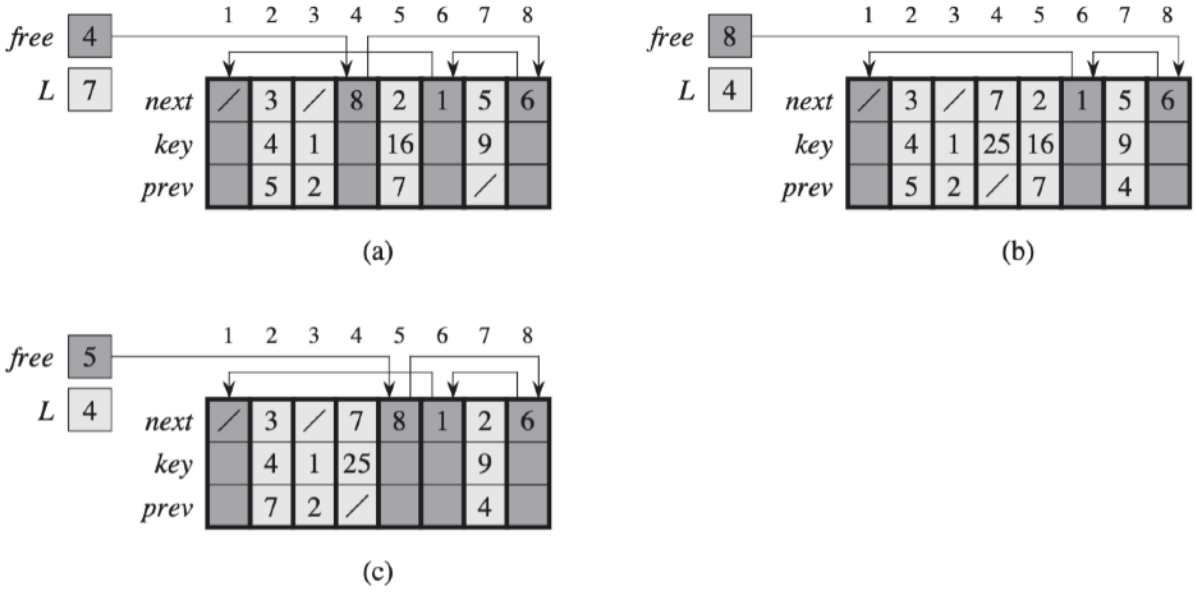


Each element in list is an object that takes up a subarray of length 3 inside the array. Key, next, prev

Allocating and freeing objects

Some systems have **garbage collector** which determines which objects are unused.

We can have a free list and list L. The free list is like a stack where the last object allocated is the last one freed. A object can not be in both free list and list L at the same time.



(a)In the free stack, 4 is free so the next 25 will be allocated to 4. (b) L is now 4 since 25 was inserted into 4. Now 8 is up next in free stack. (c)16 is deleted from 5 so 5 is empty. Now 5 is added to the free stack and is up next to allocate object.

Pseudo code

allocateObject() {

If (free == null) println("out of space")

Else {free = x.next; return x}

}

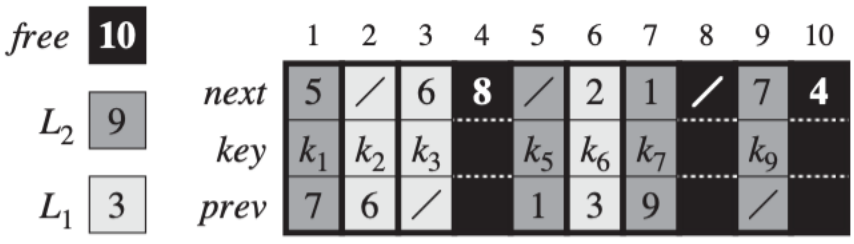
freeObject(x) {

x.next = free

Free = x

}

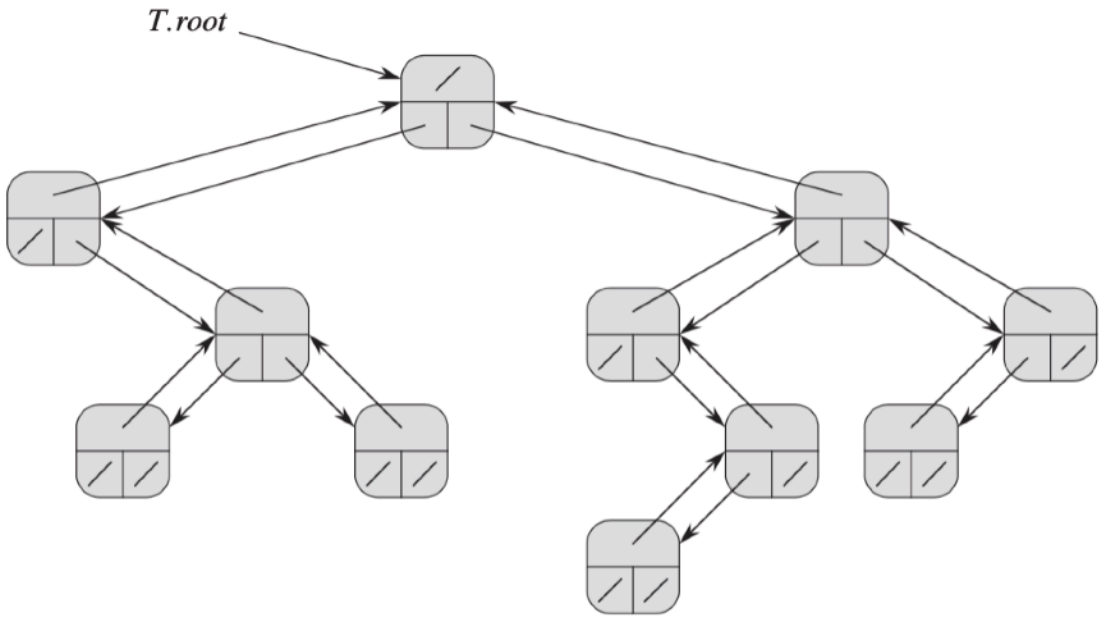
Two linked list

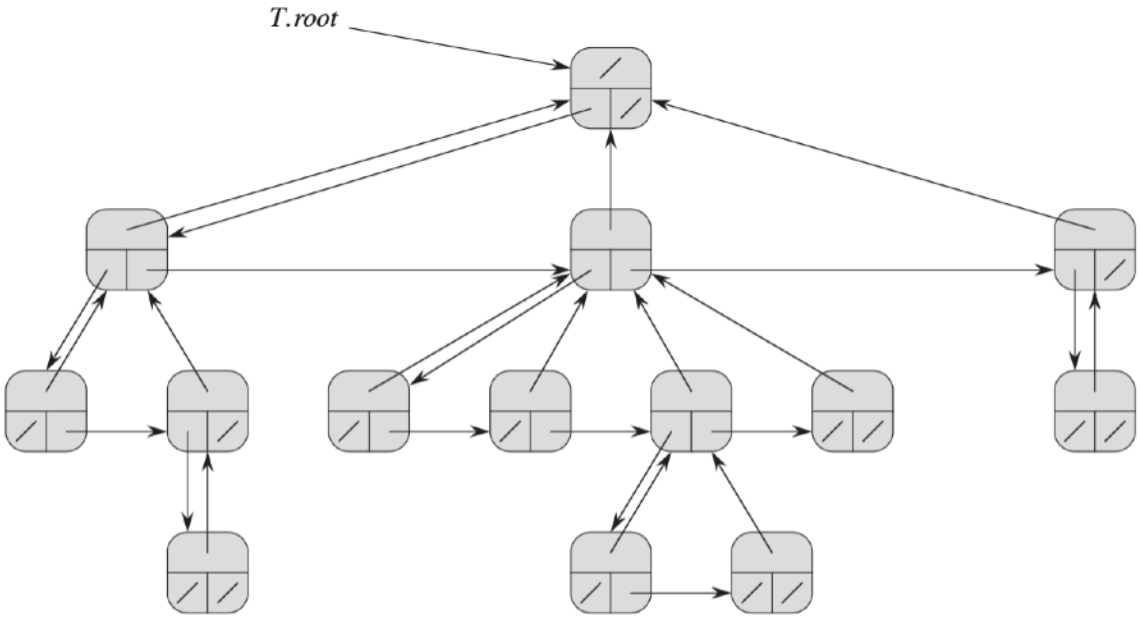


Rooted trees

Trees are represented by root and children . When the root is empty, the tree is empty. Each node has only two pointers. One points to left child and the other points to right sibling.

Binary Tree





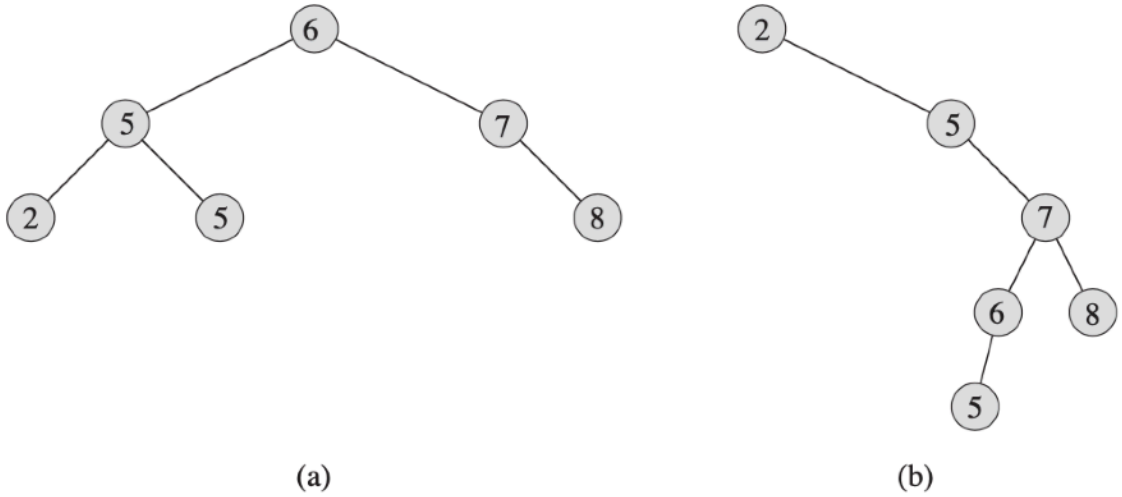
Left-child, right-sibling

Hash Tables

- Hash tables are extremely efficient average case, O(1) but worst case can be theta(n).
- Perfect hashing is able to support O(1) queries for worst case.
- Hash tables generally use array proportion to the total number of keys stored. The array index is computed from the key

Binary Search Tree

Binary tree run in theta(lgn) but it can run in theta(n) if the tree is just a linear chain of nodes. Although there is no guarantee, Red-Black trees and B-trees generally fixes this issue.



(a) (b)

(a) and (b) both have 6 nodes but (a) is more efficient because it has a height of 2 compared to a height of 4 for (b)

binary-search-tree property:

> Let x be a node in a binary search tree. If y is a node in the left subtree of x, then $y.key \leq x.key$. If y is a node in the right subtree of x, then $y.key \geq x.key$.

3 ways to traverse a binary search tree:

1. Inorder: left, parent, right
2. Preorder: parent, left, right
3. Postorder: left, right, parent

Search: Searches if node exists in bst. space and time runs in proportion to height O(h). Worst case can be O(n)

Min: finds min value in bst

Max: finds max value in bst

Insert: inserts into bst by compare its value with existing nodes

Successor: finds the successor to the node

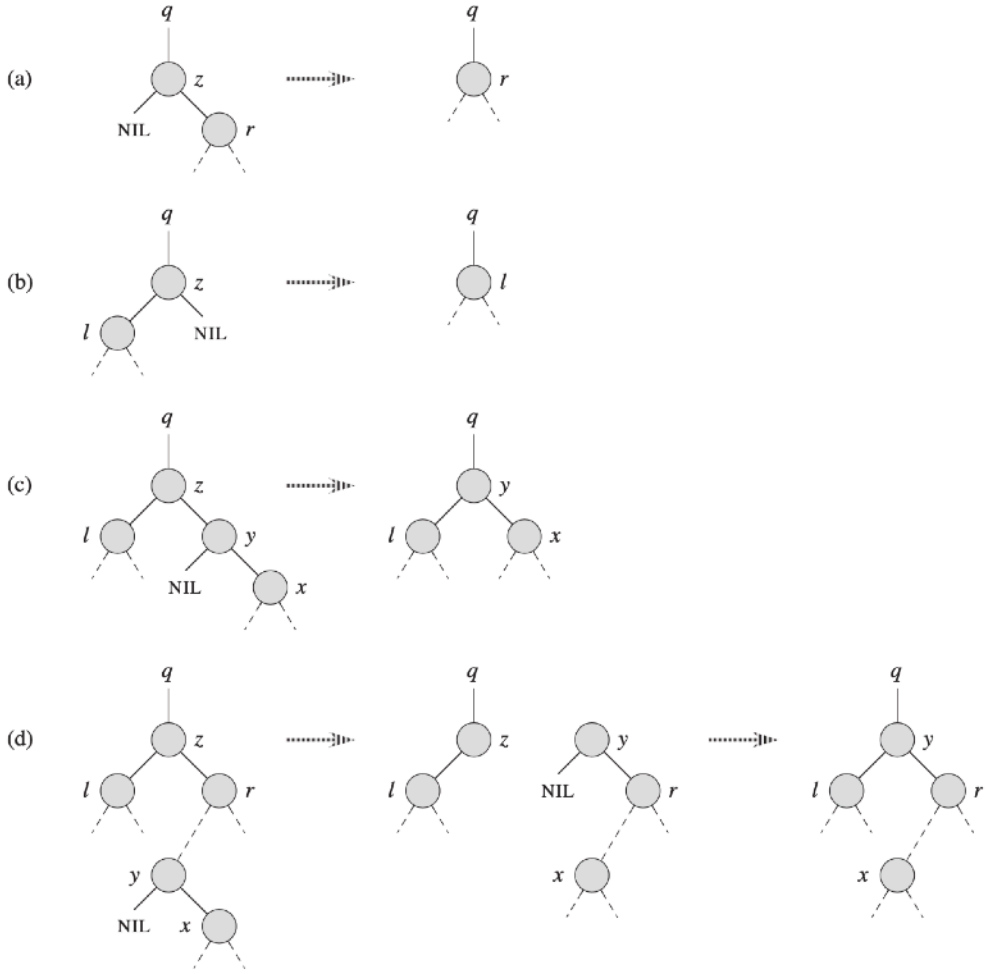
Transplant: replaces the subtree at node u with the subtree at node v

Delete:

Cases to consider:

1. If node z has no children, we can simply remove the root node
2. If node z has one children, simply replace the node z position with z's child
3. If node z has two children, find z's successor y. Y replaces z's position and the rest of z's left subtree becomes y's new left subtree. The rest of z's right subtree becomes y's new right subtree.

Insertion & deletion diagram:



(a) z is deleted and r becomes the new root (1 child right)

(b) z is deleted and l becomes the new root (1 child left)

(c) z is melted and y becomes the new root.