Course: Data Structures

Instructors: Kshitiz Verma & Sonam Nahar

The LNMIIT Jaipur

February 27, 2014

Encounters

Lectures

- L1
- Monday 11:00-12:00 (A) 16:00-17:00 (B)
- Tuesday 10:00-11:00 (A) 15:00-16:00 (B)
- Wednesday 09:00-10:00 (A) 14:00-15:00 (B)

I can be reached via email at

cse332instructor@gmail.com

Please join the following google group for regular updates

https://groups.google.com/forum/#!forum/ds-Inmiit-2013

Still not happy??

Drop in my office at any time
Room No. 262 (In front of Server Room)

Grading Scheme

Load distribution

- Total credits = 5
- 20% Midsem I
- 20% Midsem II
- 40% Endsem
- 20% Lab exercises (Roughly 2% each lab)

Exam rules

- Exams will be CLOSED book and CLOSED notes
- In my part, you can expect at least 60% directly from what is taught in the classes (So pay attention in the class)
- Questions will be of analytical and design nature, i.e., you MUST UNDERSTAND what you are learning

References

- Introduction to Algorithms, by Thomas H. Coreman et al. MIT Press.
-

Computational Complexity of Programs

Complexity

- What is it? Why is it important?
- Space Complexity
- Is computer infinitely powerful?
- Programs may run for centuries!!!

Complexity comparison

- Merge sort
- Bogosort

Speed up of programs

Due to algorithm change

Data Structures

Data Structure: Our focus

- Way to store and organize data
- Facilitate access and modifications
- No general purpose data structure
- Crucial to understand when to apply which one

Example

- Array and trees for searching
- Dictionary lookup vs randomly organized

Speed up of programs

- Also depends on which data structure is used
- Today we analyze growth of functions

Growth of functions

Asymptotic Efficiency of algorithms

- Just the highest order term
- Impact of others become small as "n" grows larger
- Enough to provide relative comparison

Basic Asymptotic Notations

- O Notation
- Θ Notation
- Ω Notation

O Notation– Asymptotic Upper Bound

- For a given function $g(n)$, we denote by the set of functions
- $O(g(n)) = \{f(n)$: there exist positive constants c and n_0 such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0\}$.
- Upper bound on a function within a constant factor
- $f(n) = O(g(n))$ to indicate that a function $f(n)$ is a member of the set $O(g(n))$
- Strange but $n = O(n^2)$

Ω Notation– Asymptotic Lower Bound

- For a given function $g(n)$, we denote by $\Omega(g(n))$ the set of functions
- $\Omega(g(n)) = \{f(n)$: there exist positive constants c and n_0 such that $0 \leq cg(n) \leq f(n)$ for all $n \geq n_0\}$.

Θ Notation

- For a given function $g(n)$, we denote by $\Theta(g(n))$ the set of functions
- $\Theta(g(n)) = \{f(n)$: there exist positive constants c_1, c_2, and n_0 such that $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq n_0\}$
- For any two functions $f(n)$ and $g(n)$, we have $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$

```
BUBBLESORT(A)
1 for i ← 1 to length[A]
2     do for j ← length[A] downto i + 1
3            do if A[j] < A[j - 1]
4                   then exchange A[j] ↔ A[j - 1]
```

```
DFS(G)
1  for each vertex u □ V [G]
2       do color[u] ← WHITE
3          π[u] ← NIL
4  time ← 0
5  for each vertex u □ V [G]
6       do if color[u] = WHITE
7             then DFS-VISIT(u)
DFS-VISIT(u)
1  color[u] ← GRAY     ▹White vertex u has just been discovered.
2  time ← time +1
3  d[u] time
4  for each v □ Adj[u]  ▹Explore edge(u, v).
5       do if color[v] = WHITE
6             then π[v] ← u
7                          DFS-VISIT(v)
8  color[u] BLACK      ▹ Blacken u; it is finished.
9  f [u] ▹ time ← time +1
```

```
BFS(G, s)
 1  for each vertex u □ V [G] - {s}
 2       do color[u] ← WHITE
 3          d[u] ← ∞
 4          π[u] ← NIL
 5  color[s] ← GRAY
 6  d[s] ← 0
 7  π[s] ← NIL
 8  Q ← ∅
 9  ENQUEUE(Q, s)
10  while Q ≠ ∅
11      do u ← DEQUEUE(Q)
12         for each v □ Adj[u]
13             do if color[v] = WHITE
14                   then color[v] ← GRAY
15                        d[v] ← d[u] + 1
16                        π[v] ← u
17                        ENQUEUE(Q, v)
18         color[u] ← BLACK
```

Introduction to Data Structures

- Sets are important in computer science too
- Sets manipulated by algorithms may grow, shrink or change over times
- Such sets are dynamic
- To support above operations, insert/delete elements of such set
- In a typical implementation, each element is represented by an object whose fields can be examined and manipulated if we have pointer to the object

Operations on dynamic sets

- Operations on dynamic sets can be classified as queries and modifying operations

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 $key[x] = k$, 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 fields 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 uses 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.

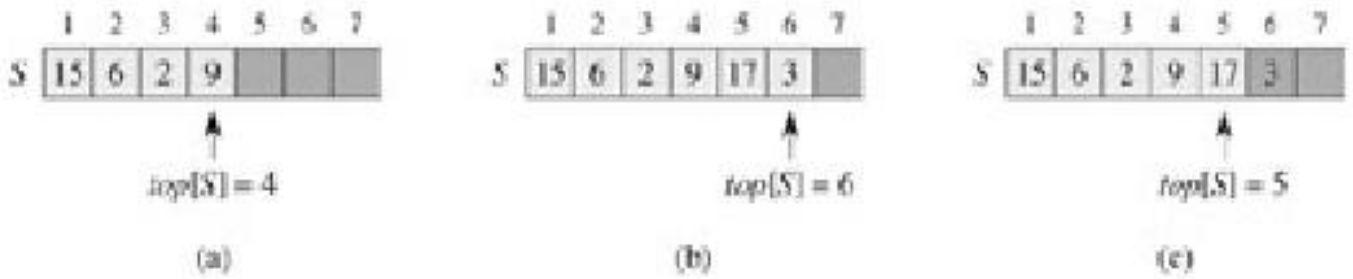
Elementary Data Structures

- Arrays
- Stacks
- Queues
- Linked Lists
- Trees

- Elements removed from the set by the DELETE operation is prespecified
- Element deletion follows last in, first out, or LIFO policy
- INSERT operation on a stack is often called PUSH
- DELETE operation (does not take an argument) is often called POP
- Example: Plates in cafeterias

Stacks
Implementing Stacks

- Stack of n elements as an array
- The array has attribute *top*[*S*], indexes most recently inserted element



- When *top*[*S*]=0, stack is empty
- If an empty stack is POPed, it *underflows*
- If *top*[*S*] exceeds *n*, it *overflows*

Implementing stack

```
STACK-EMPTY(S)
1  if top[S] = 0
2       then return TRUE
3       else return FALSE

PUSH(S, x)
1  top[S] ← top[S] + 1
2  S[top[S]] ← x

POP(S)
1  if STACK-EMPTY(S)
2     then error "underflow"


3     else top[S] ← top[S] - 1
4          return S[top[S] + 1]
```

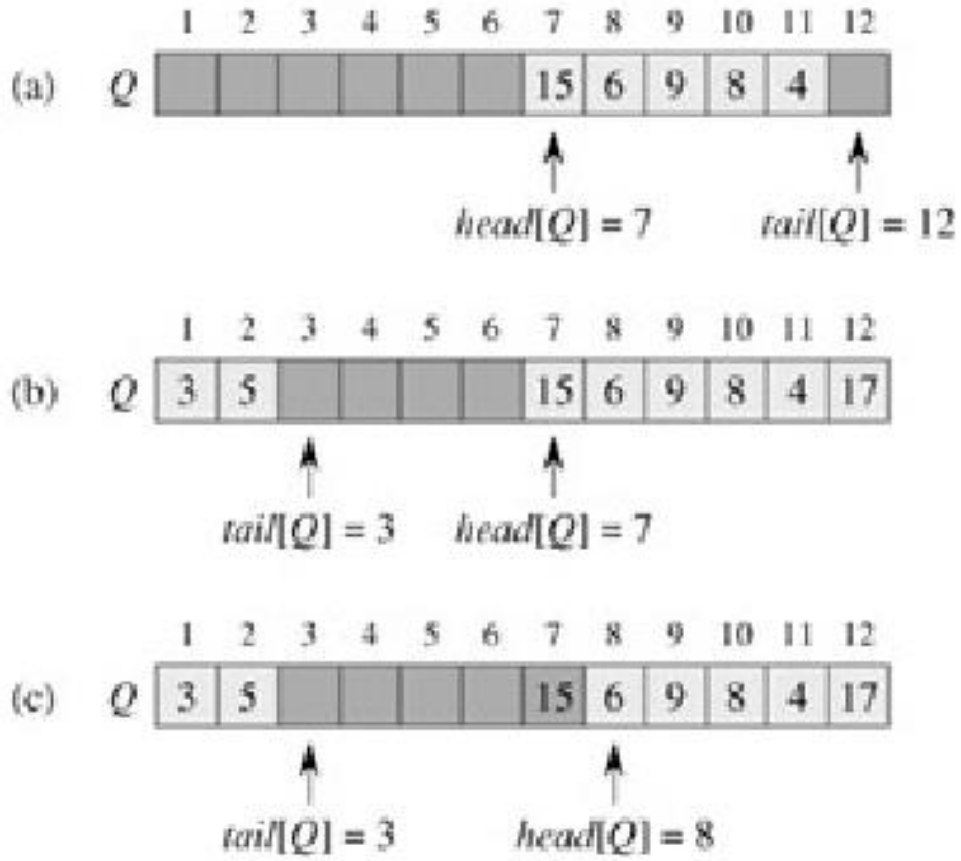
Queues

- INSERT operation on a queue: ENQUEUE
- DELETE operation on a queue: DEQUEUE (takes argument?)
- FIFO: Has a head and a tail
- ENQUEUE: Takes place at the tail of the queue
- DEQUEUE: The head of the queue

Queues

- At most $n - 1$ elements
- $head[Q]$ indexes head
- $tail[Q]$ indexes the next location at which a newly arriving element will be inserted
- $head[Q]$, $head[Q] + 1$,..., $tail[Q] - 1$
- Circular queue
- If queue is empty, $head[Q] = tail[Q]$
- Initially, $head[Q] = tail[Q] = 1$
- Underflow, overflow conditions

Queue Implementation



Implementing queue

```
ENQUEUE(Q, x)
1   Q[tail[Q]] ← x

2   if tail[Q] = length[Q]
3      then tail[Q] ← 1
4      else tail[Q] ← tail[Q] + 1

DEQUEUE(Q)
1   x ← Q[head[Q]]
2   if head[Q] = length[Q]
3      then head[Q] ← 1
4      else head[Q] ← head[Q] + 1
5   return x
```

Linked Lists

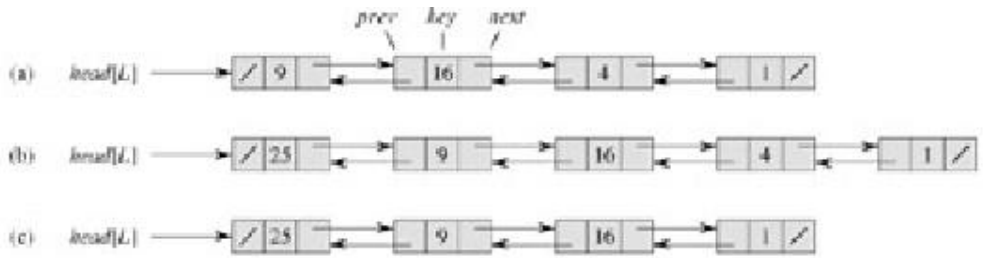
- Objects are arranged in a linear order
- Unlike array, the order in a linked list is determined by a pointer in each object
- Singly, doubly, circular linked lists
- Each element of doubly linked list L is an object with a *key* field and two pointer fields: *next* and *prev*
- Given an element x, $next[x]$ points to its successor, $prev[x]$ to its predecessor
- What if $prev[x] = NIL$? $next[x] = NIL$?
- $head[L]$ points to the first element of L
- What if $head[L] = NIL$?

Types

- Singly linked: Omit the *prev* pointer in each element
- Sorted: The minimum element is the head of the list, and maximum is the tail
- Circular: *prev* of the head points to the tail, and the *next* of the tail points to the head
- We assume unsorted and doubly linked

Linked Lists

- Doubly linked list L representing the dynamic set {1,4,9,16}
- *next* field of the tail and *prev* field of the head are *NIL*
- LIST-INSERT(L,x), where *key*[*x*]=25, leads to new head that points to old object with key 9
- LIST-DELETE(L,x), where *x* points to the object with key 4



Searching a linked list

- LIST-SEARCH(L,k) finds the first element with key k in list L
- Simple linear search, returns pointer to the element
- What do LIST-SEARCH(L,4) & LIST-SEARCH(L,7) return?

```
LIST-SEARCH(L, k)
1  x ← head[L]
2  while x ≠ NIL and key[x] ≠ k
3        do x ← next[x]
4  return x
```

Inserting into a linked list

- Given an element x whose key field has already been set

```
LIST-INSERT(L, x)
1  next[x] ← head[L]
2  if head[L] ≠ NIL
3     then prev[head[L]] ← x
4  head[L] ← x
5  prev[x] ← NIL
```

- Where does it insert?

Deleting from a linked list

- Removes an element x from L, given a pointer to x
- To delete an element with a given key, LIST-SEARCH is called to retrieve a pointer to the element

```
LIST-DELETE(L, x)
1  if prev[x] ≠ NIL
2       then next[prev[x]] ← next[x]
3       else head[L] ← next[x]
4  if next[x] ≠ NIL
5       then prev[next[x]] ← prev[x]
```

Sentinels

- A dummy object that allows to simplify boundary conditions
- For example, suppose we have list L
- And an object $nil[L]$ that represents NIL but has all the fields of the other list elements
- Wherever we have a reference to NIL in list code, we replace it by a reference to the sentinel $nil[L]$
- Read more on page 180 of the book

Representing Rooted Trees

- Using linked lists
- Represent each node of a tree by an object
- As with linked lists, each node contains a *key* field

Binary Trees

- Use fields *p*, *left* and *right* to store pointers to the parent, left child and right child of each node
- If $p[x]$=NIL, x is the root
- If node x has no child, then $left[x]$=NIL
- The root of the entire tree T is pointed to by the attribute $root[T]$
- If $root[T]$=NIL, then the tree is empty

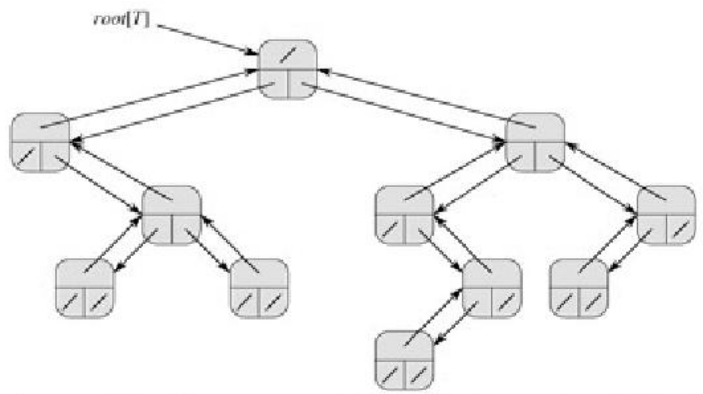


Figure 10.9: The representation of a binary tree T. Each node x has the fields $p[x]$ (top), $left[x]$ (lower left), and $right[x]$ (lower right). The *key* fields are not shown.

Binary Search Trees

- Support many dynamic-set operations including SEARCH, MINIMUM, MAXIMUM, PREDECESSOR, SUCCESSOR, INSERT and DELETE
- Binary Search Tree (BST)
- Can be represented by linked data structure
- Each node constains, key, left, right and p that points to left child, right child and parent respectively
- The root node is the only node in the tree whose parent field is NIL

Binary Search Trees

The keys in a BST are always stored to satisfy **BST property**

BST property

- Let x be a node in a BST.
- If y is a node in the left subtree of x, then $key[y] \leq key[x]$
- If y is a node in the right subtree of x, then $key[x] \leq key[y]$

Binary Search Trees

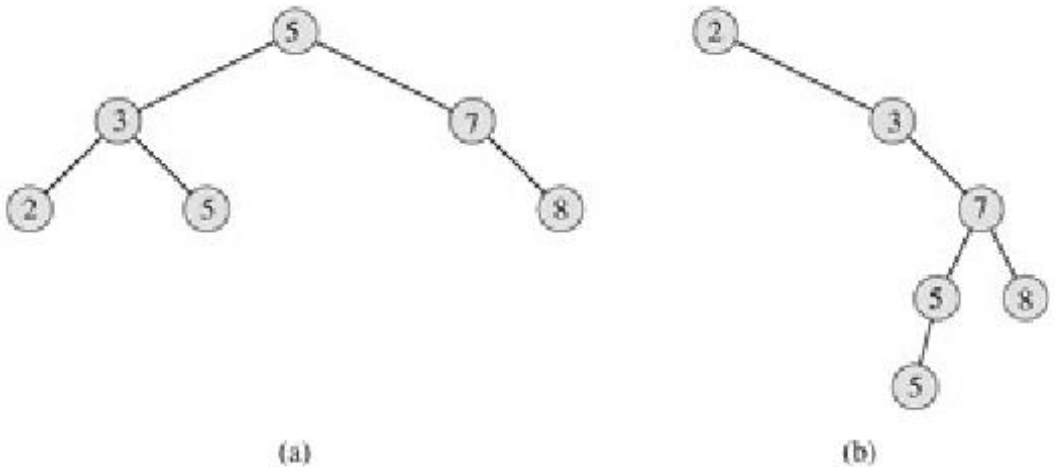


Figure 12.1: Binary search trees. For any node x, the keys in the left subtree of x are at most $key[x]$, and the keys in the right subtree of x are at least $key[x]$. Different binary search trees can represent the same set of values. The worst-case running time for most search-tree operations is proportional to the height of the tree. *(a)* A binary search tree on 6 nodes with height 2. *(b)* A less efficient binary search tree with height 4 that contains the same keys.

Inorder Tree Walk

Call INORDER-TREE-WALK(root[T])

```
INORDER-TREE-WALK(x)
1   if x ≠ NIL
2       then INORDER-TREE-WALK(left[x])
3            print key[x]
4            INORDER-TREE-WALK(right[x])
```

Searching a BST

Given a pointer to the root of the tree and a key k

```
TREE-SEARCH (x, k)
1   if x= NIL or k = key[x]
2       then return x
3   if k < key[x]
4       then return TREE-SEARCH(left[x], k)
5       else return TREE-SEARCH(right[x], k)
```

An example

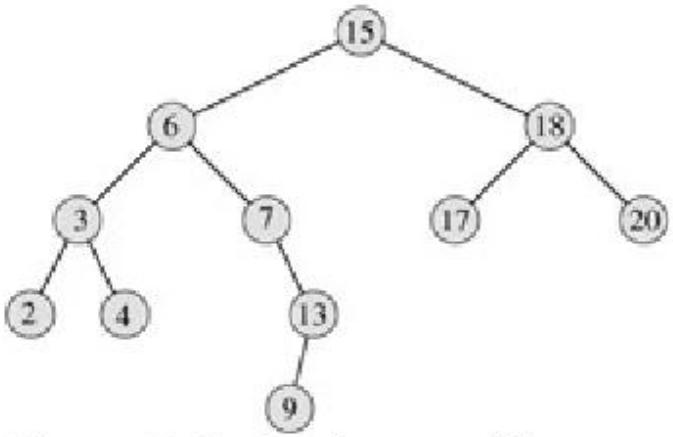


Figure 12.2: Queries on a binary search tree. To search for the key 13 in the tree, we follow the path $15 \rightarrow 6 \rightarrow 7 \rightarrow 13$ from the root. The minimum key in the tree is 2, which can be found by following *left* pointers from the root. The maximum key 20 is found by following *right* pointers from the root. The successor of the node with key 15 is the node with key 17, since it is the minimum key in the right subtree of 15. The node with key 13 has no right subtree, and thus its successor is its lowest ancestor whose left child is also an ancestor. In this case, the node with key 15 is its successor.

Alternative search algorithm

```
ITERATIVE-TREE-SEARCH(x, k)
1  while x ≠ NIL and k ≠ key[x]
2        do if k < key[x]
3              then x ← left[x]
4              else x ← right[x]
5  return x
```

Its complexity?

Minimum and maximum

```
TREE-MINIMUM (x)
1  while left[x] ≠ NIL
2        do x ← left[x]
3  return x
```

```
TREE-MAXIMUM(x)
1  while right[x] ≠ NIL
2        do x ← right[x]
3  return x
```

Successor

- Find successor in the sorted order determined by an inorder tree walk
- If all keys are distinct, the successor of a node x is the node with the smallest key greater than $key[x]$

```
TREE-SUCCESSOR(x)
1  if right[x] ≠ NIL
2       then return TREE-MINIMUM (right[x])
3  y ← p[x]
4  while y ≠ NIL and x = right[y]
5       do x ← y
6          y ← p[y]
7  return y
```

Insertion

- Insertion should be such that the BST property is maintained
- To insert a new value v into a BST T
- Node z for which $key[z] = v$, $left[z] = NIL$, and $right[z] = NIL$

```
TREE-INSERT(T, z)
 1  y ← NIL
 2  x ← root[T]
 3  while x ≠ NIL
 4        do y ←  x
 5           if key[z] < key[x]
 6              then x ← left[x]
 7              else x ← right[x]
 8  p[z] ← y
 9  if y = NIL
10     then root[T] ← z                ⊹ Tree T was empty
11     else if key[z] < key[y]
12             then left[y] ← z
13             else right[y] ← z
```

Insertion: An example

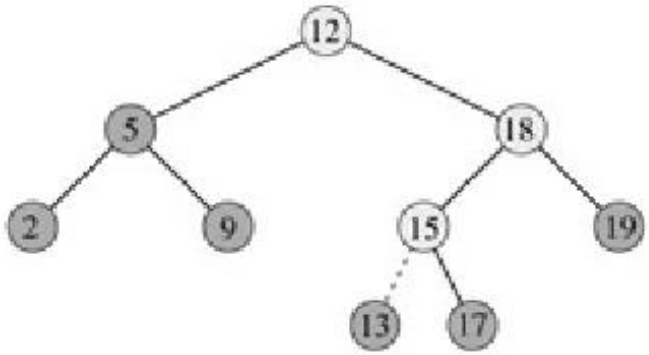


Figure 12.3: Inserting an item with key 13 into a binary search tree. Lightly shaded nodes indicate the path from the root down to the position where the item is inserted. The dashed line indicates the link in the tree that is added to insert the item.

Deletion

- Deletion should be such that the BST property is maintained
- Argument is a pointer to the node z (to be deleted)
- Three cases

Deletion Cases

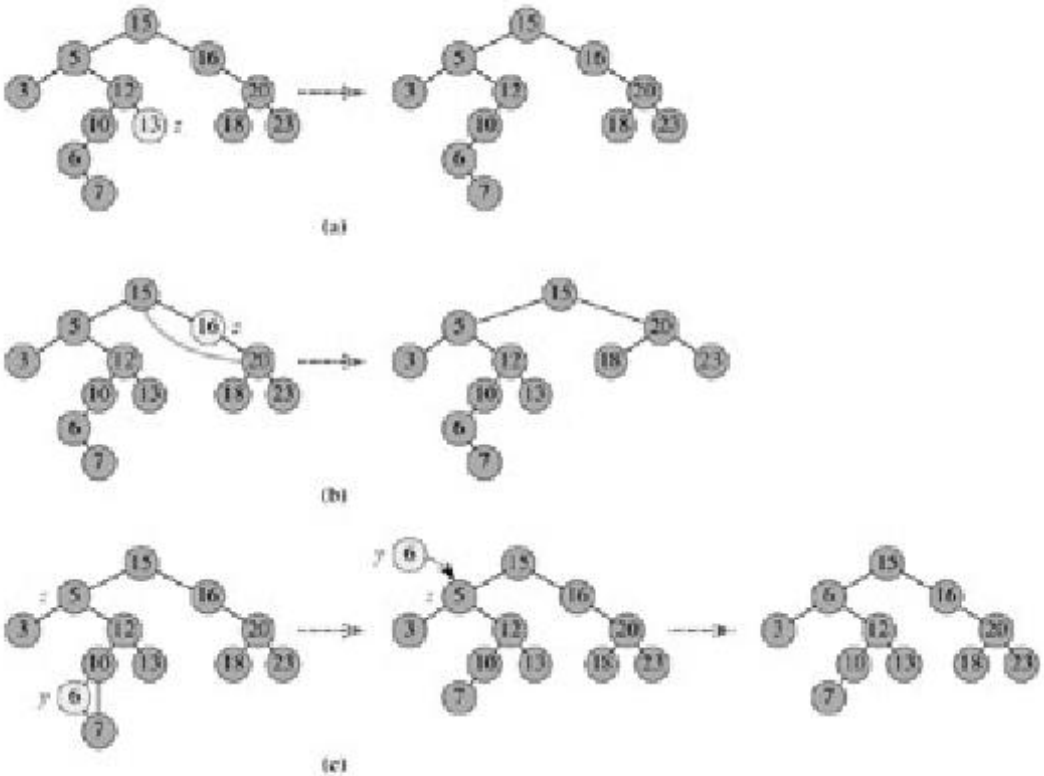


Figure 12.4: Deleting a node z from a binary search tree. Which node is actually removed depends on how many children z has; this node is shown lightly shaded. *(a)* If z has no children, we just remove it. *(b)* If z has only one child, we splice out z. *(c)* If z has two children, we splice out its successor y, which has at most one child, and then replace z's key and satellite data with y's key and satellite data.

Deletion

```
TREE-DELETE(T, z)
 1  if left[z] = NIL or right[z] = NIL
 2       then y ← z
 3       else y ← TREE-SUCCESSOR(z)
 4  if left[y] ≠ NIL
 5       then x ← left[y]
 6       else x ← right[y]
 7  if x ≠ NIL
 8       then p[x] ← p[y]
 9  if p[y] = NIL
10       then root[T] ← x
11       else if y = left[p[y]]
12                 then left[p[y]] ← x
13                 else right[p[y]] ← x
14  if y ≠ z
15       then key[z] ← key[y]
16            copy y's satellite data into z
17  return y
```

Feedback

- Examples of real world and more examples
- Focus on how to code these algorithms
- Assignments please!!! / Pratice questions
- Fast pace!
- It would be better if we get slides in advance so that we can come prepare for next class.
- only thing that is still a mystery is ,what are we going to extract out of this course !
- proper notes should be maintained in the class. But right now ..no notes are available to us.

Feedback

- The lectures must be prepared in a more coherent manner. Pending explanations should not be postponed by more than one class. Also, apart from just displaying pseudocodes in front of us, an attempt must be made to teach us how to write those pseudocodes- it is that part which is really hard.
- Sir you yourself get confused sometimes ..you have the knowledge but you are not able to give the same to students i feel that ! we are not able to relate the things we study in class with the things which we'll be doing in the lab !! try to elaborate more and give more examples on complexity !
- Sir, explanations are not quite convincing. So, the topic remains weak.

Solving recurrences

The Master Method

- Used for recurrences of the form $T(n) = aT(n/b) + f(n)$
- $a \geq 1, b \geq 1$ are constants, $f(n)$ is asymptotically positive function
- Problems of size n are divided into a problems of size n/b
- $f(n)$ represents the cost of dividing the problem and combining the results of the subproblem + everything else
- $T(n)$ can be bounded asymptotically as follows:
  1. If $f(n) = O(n^{log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{log_b a})$
  2. If $f(n) = \Theta(n^{log_b a})$, then $T(n) = \Theta(n^{log_b a} logn)$
  3. If $f(n) = \Omega(n^{log_b a + \epsilon})$ for some constant $\epsilon > 0$ **AND** if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and sufficiently large n, then $T(n) = \Theta(f(n))$

Examples

1. $T(n) = 9T(n/3) + n$
2. $T(n) = 9T(2n/3) + 1$
3. $T(n) = 3T(n/4) + nlogn$
4. $T(n) = 2T(n/2) + nlogn$
5. $T(n) = 2T(n/2) + n$
6. $T(n) = 2T(n/2) + n^2$
7. $T(n) = 2T(n/2) + n^3$
8. $T(n) = 4T(n/2) + n^2 logn$
9. Compute binary search recurrence and complexity

Hint: Compute $n^{log_b a}$ and $f(n)$ for each of them, then see which case applies

AVL Trees

For today's lecture refer to "Data Structures and Algorithm Analysis in C" by Mark Allen Weiss.

- An AVL tree is a BST with a balance condition
- The balance condition must be "easy" to maintain
- Ensures that the depth of the tree is O(logN)
- The simplest idea requires that the left and right subtrees have the same height
- AVL tree is identical to a BST, except that for **every** node in the tree, the height of the left and right subtrees can differ by at most 1
- The height of an empty tree is **defined** to be -1

AVL Trees

- Height information is kept for each node
- The height of an AVL tree is O(logN)
- Impact on complexity?
- All the tree operations can be performed in O(logN), except possibly insertion
- The inserting node could violate the AVL tree property
- *Rotation* is used to restore AVL tree property

Insertion in AVL trees

- After an insertion, nodes that are on the path from the insertion point to the root might have their balance altered
- Only those nodes have their subtrees altered
- Follow the path up to the root to update balancing information
- Violation might occur in four cases
- If the node to be rebalanced is α
  1. An insertion into the left subtree of the left child of α
  2. An insertion into the right subtree of the left child of α
  3. An insertion into the left subtree of the right child of α
  4. An insertion into the right subtree of the right child of α
- In lab, implement a doubly linked list of size 8

B-Trees

- Search tree that is not a BST
- A B-tree of order M is a tree with the following structural properties

1. The root is either a leaf or has between 2 and M children
2. All nonleaf nodes (except the root) have between $\lceil \frac{M}{2} \rceil$ and M children
3. All leaves are at the same depth

B-Trees

- All data are stored at the leaves
- Contained in each interior node are pointers $P_1, P_2, ..., P_M$ to the children, and values $k_1, k_2, .., k_{M-1}$ representing the smallest key found in the subtrees $P_2, P_3, ..., P_M$ respectively
- For every node, all the keys in subtree P_1 are smaller than the keys in subtree P_2, and so on
- The leaves contain all the actual data, which are either the keys themselves or pointers to records containing the keys
- Search, insertion, deletion, sequential access in logarithmic time

Binary Heaps

- Like BSTs, heaps have two properties
- A structure property and a heap order property
- Like AVL trees, operation on a heap may destroy properties
- So a heap operation continues until properties are in order
- A heap is a binary tree that is completely filled
- A possible exception is the bottom level, which is filled from left to right
- Known as complete binary tree
- Height h is in between 2^h and $2^{h+1} - 1$ nodes
- One to one correspondence between an array and a heap is easy

Structure property

- Array A that represents heap is an object with two attributes
- Length[A] (No. of elements in the array) and heap-size[A] (the number of elements in the heap stored within an array A)
- No element past A[heap-size[A]], where heap-size[A]$\leq$ length[A], is an element of the heap
- The root of the heap is A[1], and given index i of the node

```
PARENT(i)
    return ⌊i/2⌋

LEFT(i)
    return 2i

RIGHT(i)
    return 2i + 1
```

Structure property

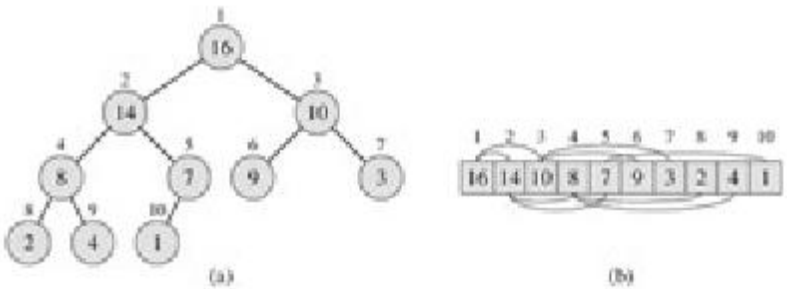


Figure 6.1: A max-heap viewed as *(a)* a binary tree and *(b)* an array. The number within the circle at each node in the tree is the value stored at that node. The number above a node is the corresponding index in the array. Above and below the array are lines showing parent-child relationships; parents are always to the left of their children. The tree has height three; the node at index 4 (with value 8) has height one.

Heap Order Property

- The goal is to find the extreme element quickly, so the smallest/largest element should be at the root

Max Heap Property

- $A[parent(i)] \geq A[i]$

Min Heap Property

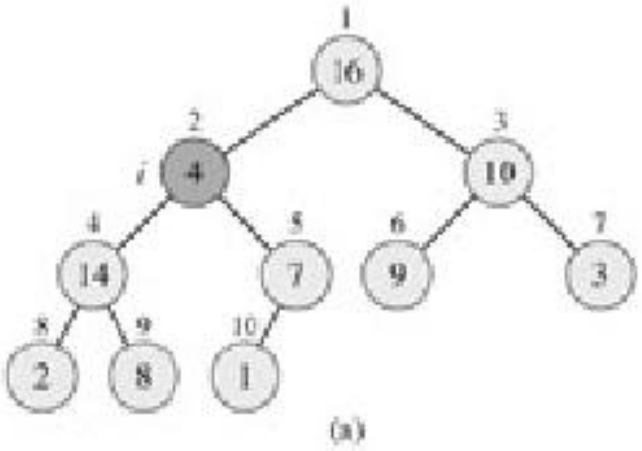
- $A[parent(i)] \leq A[i]$

Maintaining the heap property

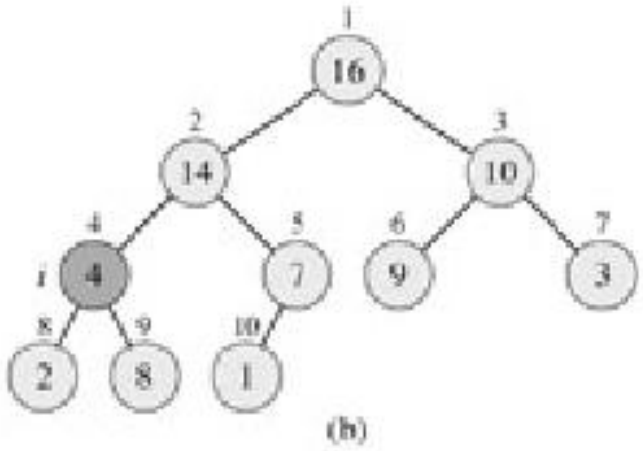
- Inputs are array A and an index into the array
- MAX-HEAPIFY assumes that the binary trees rooted at LEFT(i) and RIGHT(i) are max heaps

```
MAX-HEAPIFY(A, i)
 1 l ← LEFT(i)
 2 r ← RIGHT(i)
 3 if l ≤ heap-size[A] and A[l] > A[i]
 4     then largest ← l
 5     else largest ← i
 6 if r ≤ heap-size[A] and A[r] > A[largest]
 7     then largest ← r
 8 if largest ≠ i
 9     then exchange A[i] ↔ A[largest]
10          MAX-HEAPIFY(A, largest)
```

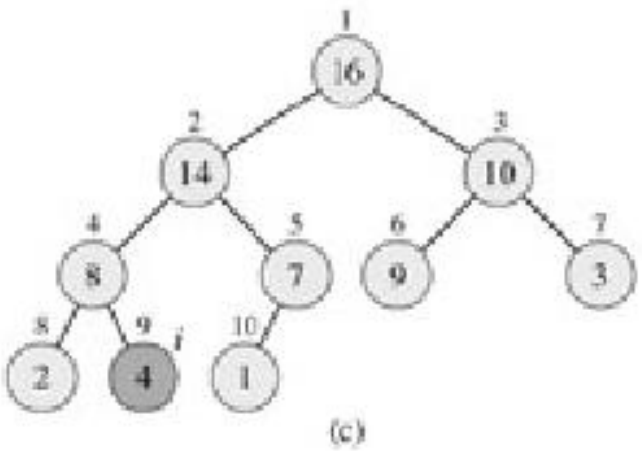
Max Heapify



(a)



(b)



(c)

Build Max Heap

Convert an array A into a max heap

```
BUILD-MAX-HEAP(A)
1  heap-size[A] ← length[A]
2  for i ← ⌊length[A]/2⌋ downto 1
3       do MAX-HEAPIFY(A, i)
```

Pre-order and post-order tree traversal if we get time

Polynomial representation

- A single variable polynomial can be generalized as

$$f(x) = \sum_{i=0}^{n} a_i x^i$$

  which can be expanded to

$$a_n x^n + a_{n-1} x^{n-1} + ... + a_1 x^1 + a_0 x^0$$

- Two data: Exponent(integer) and Coefficient(real)
- Order of the polynomial: the highest exponent
- Common operations on a polynomial
  - Add, subtract, multiply, differentiate, integrate, etc
- How to implement? Which data structure to use?
  - Consider efficiency in space and time

Array Implementation

- Consider $f(x) = 8x^5 + 4x^3 + 3x + 2$
- Index represents exponent
- The value in the index can represent the corresponding coefficient
- Consider another polynomial $f(x) = 5x^{974} + 3x^{23} + x + 12$
- Advantages of using an array
  - Ease of storage and retrieval
- Disadvantages
  - Allocate size ahead of time, array size may be large for sparse polynomials

Double Array Implementation

- Two arrays: One for exponents and the other for coefficients
- Advantages of using a double array
  - Saves space
- Disadvantages
  - Allocate size ahead of time
  - Two arrays instead of just one (added complexity)

Linked List Implementation(Singly)

- A node has two fields: exponents and coefficients
  - How?
- Advantages of linked list implementation
  - Saves space and easy to maintain
  - Dynamic allocation possible
- Disadvantages
  - Can't go backwards through the lists
- Homework: Write an algorithm to add two polynomials in a linked list implementation
  - With and without Horner's rule
  - High probability that I may ask this question in exam

Matrix representation

- How to represent a matrix?
  - $n \times m$ So a 2D array
- What if matrix is sparse?
  - i.e., only few elements are non zero
  - Example: Web page matrix
- Each non zero element can be represented by a triple
  - (row, column, value)
  - How to implement this triple? Array or linked list?
  - Array linear list representation
  - Linked list: Row, column, value, next

Applications of stacks
Balancing Symbols

- Check whether everything is balanced
  - Brace, bracket, parentheses
  - The sequence [()] is legal, but [(]) is wrong

The Algorithm

Make an empty stack. Read characters until end of file. If the character is an opening symbol, push it onto the stack. If it is a closing symbol, then if the stack is empty, report an error. Otherwise, pop the stack. If the symbol popped is not the corresponding opening symbol, then report an error. At the end of file, if the stack is not empty, report an error.

Applications of stacks
Evaluation of arithmetic expressions

- Consider evaluating the expression 2 * 3 + 3 + 5 * 2
  - Multiply 2 and 3. Store the result in A.
  - Add 3 to A. Store the result in A.
  - Multiply 5 and 2. Store the result in B.
  - Add A and B.
- Above sequence of operations can be written as:
- 2 3 * 3 + 5 2 * +
- This notation is called **postfix**

Types of notations for expressions

- Infix: Operator is between operands A+B
- Postfix: Operator follows operands AB+
- Prefix: Operator precedes operands +AB
- All expressions cannot be evaluated by using left to right order
- Operators in a postfix expression are always in the correct evaluation order
- Thus, evaluation -> first conversion and then evaluation of postfix expression
- Use of stacks in both the processes

Infix to postfix conversion

- a + b * c (precedence of * is higher than +)
- a + (b * c) (convert the multiplication)
- a + (b c *) (convert the addition)
- a (b c *) + (Remove parentheses)
- a b c * +

- Examples
  - (a + b)*c –> ab+c*
  - a+((b*c)/d) –> abc*d/+
  - (a+b)*(c-d) –> ab+cd-*
  - a-b/(c+d*e) –> abcde*+/-
  - ((a+b)*c-(d-e))/(f+g) –> ab+c*de- -fg+/

Evaluating a postfix expression
Algorithm

- Maintain a stack and scan the postfix expression from left to right
  - If the element is a number, push it into the stack
  - If the element is an operator #, pop twice and get A and B respectively. Calculate B#A and push it back to the stack
  - When the expression ends, the number in the stack is the final answer

Transform infix to postfix

Only operators are + -

- The order of computation depends on the precedence and order (as they appear in the expression) of operators
- left to right parsing (if precedence is same, the one on left is given higher priority)

- Algorithm
  - Maintain a stack and scan the infix expression from left to right
  - When we get a number, output it
  - When we get an operator #, pop the top element in the stack if the stack is not empty and then push(#) into the stack
  - When the expression ends, pop all the operators remain in the stack

Transform infix to postfix

Only operators are +, -, *, /

- Scan the infix expression
- left to right parsing (if precedence is same, the one on left is given higher priority)

- Maintain a stack and scan the infix expression from left to right
- When we get a number, output it
- When we get an operator #, pop the top element in the stack until there is no operator having higher priority than # and then push(#) into the stack
- When the expression ends, pop all the operators remain in the stack

Recursion

- The process of a function calling itself
- This may lead to an infinite loop
  - A common mistake in recursion
- Recursion is often expensive in terms of space
  - Recall the binary search using recursion and without recursion
- Recursion works if we can decompose a larger problem into subproblems of smaller size
- Extra class tomorrow.

Matrix Chain Multiplication
Design of algorithm

Chain of matrices

Given a sequence (chain) $A_1, A_2, ..., A_n$ of n matrices, compute the product

$$A_1 A_2 .... A_n$$

- Using the standard matrix multiplication for a pair of matrices as subroutine
- A product of matrices is **fully parenthesized** if
  - Either a single matrix
  - Or the product of two fully parenthesized matrix products, surrounded by parentheses

Matrix Chain Multiplication

- Matrix multiplication is associative, so all parenthesizations yield the same product
- For example, if the chain of matrices is A_1, A_2, A_3, A_4, the product $A_1A_2A_3A_4$ can be fully parenthesized as
  1. $(A_1(A_2(A_3A_4)))$
  2. $(A_1((A_2A_3)A_4))$
  3. $((A_1A_2)(A_3A_4))$
  4. $((A_1(A_2A_3))A_4)$
  5. $(((A_1A_2)A_3)A_4)$

Cost of multiplication of matrices

```
MATRIX-MULTIPLY(A, B)
1 if columns[A] ≠ rows[B]
2     then error "incompatible dimensions"
3     else for i ← 1 to rows[A]
4               do for j ← 1 to columns[B]
5                     do C[i, j] ← 0
6                        for k ← 1 to columns[A]
7                             do C[i, j] ← C[i, j] + A[i, k] · B[k, j]
8          return C
```

- Cost of multiplying two matrices
  - Can be multiplied only if they are **compatible**
  - If A is $p \times q$ and B is $q \times r$, the resulting matrix C is $p \times r$
  - Cost = Number of scalar multiplications in line 7 is pqr

Example of different costs

- $A_1 = 10 \times 100$
- $A_2 = 100 \times 5$
- $A_1 = 5 \times 50$
- $((A_1 A_2) A_3)$ implies 5000 + 2500 = 7500 scalar multiplications
- $(A_1 (A_2 A_3))$ implies 25000 + 50000 = 75000 scalar multiplications
- An order of magnitude difference
  - Paranthesization matters!!

Matrix Chain Multiplication Problem(MCMP)

The problem statement

Given a chain $A_1, A_2, ..., A_n$ of n matrices, where for $i = 1, 2, ..., n$ matrix A_i has dimension $p_{i-1} \times p_i$, fully parenthesize the product $A_1 A_2 .. A_n$ in a way that minimizes the number of scalar multiplications.

- Note that we are not actually multiplying matrices
  - We are just looking for the best way to multiply them

Counting the number of parenthesizations

- Exhaustively checking all possible parenthesizations does not yield an efficient algorithm
- $P(n)$: Number of alternative parenthesizations of a sequence of n matrices

$$P(n) = \begin{cases} 1 & \text{if } n{=}1 \\ \sum_{k=1}^{n-1} P(k)P(n-k) & \text{if } n \geq 2 \end{cases}$$

- The solution to the above recurrence is $\Omega(2^n)$